UiO : Department of Informatics
University of Oslo

Retrieval of Genomic Data using PyTables

Henrik Glasø Skifjeld

Master's Thesis Autumn 2014

Abstract

As genomic data becomes more available, there is an increased need for analysis tools, such as the Genomic HyperBrowser. At the core of this tool is a data model known as GTrackCore – a module for storage and retrieval of genomic data. Currently, it is based on using NumPy memmaps for storage. An issue with this model is that it uses multiple files to represent a single genomic track.

The purpose of this work is to investigate whether or not the PyTables library, making use of the HDF5 data format, is suited to replace the current data model of GTrackCore. This is investigated by re-implementing large parts of GTrackCore to have it use the PyTables library and a single HDF5 file for each genomic track, as opposed to NumPy memmaps, followed by a comparison of the performance of the two versions.

This thesis primarily focuses on the retrieval part of GTrackCore, and how to utilize the advantages of the PyTables library in such a setting.

The findings shows that the cost of having a data model that stores data in a single HDF5 file makes data retrieval slower in some cases, and faster in others. Furthermore, we have seen that, as of now, GTrackCore is only facilitated to utilize a few of the advantages of the PyTables library.

Contents

List of Figures

List of Tables

Preface

Choosing a topic for my master thesis was to some extent challenging, as there is a vast area to pick from. When I was introduced to the field of bioinformatics, I was immediately intrigued. To complete my work I needed a basic understanding of genomics. This new insight has been enriching and useful.

This thesis is part of a two-folded one, where the practical work, i.e. the implementation of a new data model in a package for storage and retrieval of genetic data, is done as a collaboration by Brynjar G. Rongved and myself. In the later parts of in development of the implementation, we each chose one part of the pratical work, and the issues related to them, as the theoretical part of our theses. The storage part is covered by Rongved in [34], and the data retrieval part is the topic of this thesis.

The target audience of this thesis is master students with deep knowledge of computer science, and preferably insight into bioinformatics.

Acknowledgements

I want to thank Brynjar G. Rongved for our collaboration throughout this work, and the many hours of coding and discussions. I would also like to thank my supervisors, Geir Kjetil Sandve and Sveinung Gundersen, for academic education in the research field, interesting discussions, long programming sessions with Sveinung, and great thesis feedback.

Finally, I would like to thank my friends and family for their help and support, and especially my parents, Camilla and Knut, for their contributions.

Henrik Glasø Skifjeld
University of Oslo
August, 2014

Chapter 1

Introduction

1.1 Motivation

A substantial amount of research in the field of biology and genetics is conducted throughout the world as genomic data becomes more available and accessible. Along comes the need for efficient computational research tools. One of several tools applied in scientific work is the Genomic HyperBrowser, purposed for statistical analysis of genomic data. The HyperBrowser is written using Python, and takes advantage of the tool NumPy for efficient scientific computing. At the core of the HyperBrowser is GTrackCore – a module for storing and retrieving genomic data as efficiently as possible. It represents the data model of the Genomic HyperBrowser.

Genomic data is stored as genomic tracks, which in its simplest form is a textual file with any data related to a genome. The data is structured in columns of different data types, and separated into rows, or data lines, with a value for each of the columns. Each row is regarded a single informational unit of a genomic track, referred to as elements of the track. Since working with textual data in an analytical setting is very inefficient, the genomic tracks are often converted into binary data for faster retrieval. GTrackCore has two main functionalities: processing and storing textual genomic tracks to disk as binary data, and retrieving the processed data efficiently for analysis.

Currently, the data model stores the binary data using NumPy memmaps, with one file for each column of a genomic track, and uses an ad-hoc indexing method for fast data lookup. An issue with this data model is that filesystems has a maximum limit for number of files stored. HyperBrowser stores thousands of genomic tracks, and is nearing this limit. Moreover, having tracks stored like this makes it difficult to both distribute and share them, as opposed to dealing with a single file for each track.

Furthermore, future plans for GTrackCore involves separating it completely from HyperBrowser, and to create a separate open-source command-line tool for biological analysis on other platforms than the HyperBrowser, e.g. on home computers. For this purpose, using a standardized library for the data model may prove to be more satisfactory than using an ad-hoc one.

1.2 Goals

The main task of this work is the investigation of whether the Python library PyTables is a suitable replacement to the existing NumPy memmap solution as a new data model for GTrackCore. The PyTables library uses the HDF5 data format as an underlying storage foundation, where multiple datasets can be stored in a single HDF5 file. In addition, the PyTables library includes functionality for querying indexed data in an SQL-like fashion. To reach this goal we have re-implemented large parts of GTrackCore to operate with the new data model, and compared the two solutions to each other. The term suitable means, in this setting, that the overall performance of the new solution is somewhat equal to the old one, and that the new data model makes it possible to store genomic tracks in a single file.

The first aspect of the investigation is to find out whether using PyTables and HDF5 will solve the issues of needing multiple files to represent a single genomic track. The second is to find out if use of PyTables has a positive or negative effect on data retrieval in GTrackCore with regards to performance, and whether it is possible to further utilize the strengths of PyTables, e.g. its high-performance query engine.

This thesis will cover the second aspect of the task, i.e. the retrieval part of GTrackCore, while the first is covered by Rongved in [34].

1.3 Research questions

The research questions being investigated in this thesis is focused on the retrieval part of GTrackCore, and more specifically how to utilize the PyTables library in such a setting. While the first questions in the list below is a general one covering both parts of GTrackCore, we will mostly still focus on the retrieval part of the question.

1. Is the PyTables library, making use of the HDF5 format, a suitable replacement for NumPy memmaps in GTrackCore?
2. How is data retrieval in GTrackCore affected by the use of HDF5 and PyTables as opposed to the use of memmaps regarding performance?
3. How can the built-in indexing functionality in PyTables be used to ensure efficient data retrieval in GTrackCore?
4. Is GTrackCore facilitated to utilize the strengths of PyTables, regarding indexing and querying of tables?

Chapter 2

Background

This chapter provides the background information for this work. It is divided into three sections; biology, which is about general genetics and DNA; bioinformatics, which includes information about genomic tracks and genome analyzing tools; and lastly the technologies that are relevant to this work.

2.1 Biology

2.1.1 Genetics

The genome is the entire hereditary information of an organism, i.e. all of the genetic information as a whole. It is encoded in deoxyribonucleic acid (DNA), which is a molecule that encodes genetic instructions. To store genetic information DNA is made out of four different informational parts, called nucleotides or nucleobases. These are nitrogen-containing bases with the ability to form base-pairs with one another. The four nucleobases found in DNA are cytosine, guanine, adenine and thymine, respectively abbreviated as C, G, A and T. Each of these has a complementary nucleobase that together form a base-pair, i.e. two nucleobases that are complementary. Cytosine and guanine are complementary bases, and adenine and thymine are complementary bases. The DNA is built as a two-stranded double-helix where each position in the helix contains two nucleobases – one for each strand – that are complementary. The two strands are called sense and antisense. Genetic information can be encoded in either of the strands. The genome is comprised of long sequences of nucleobases that contain the genetic information. The nucleobases are linked together to form a directional bond, which means that a strand of DNA has a head and a tail, and consequently a direction. The genetic information is stored in one or more DNA molecules, where each is called a chromosome. Another molecule for encoding genetic material is called ribonucleic acid (RNA). In RNA the complementary base of adenine is not thymine, like in DNA, but uracil instead. The genetic material in, e.g. viruses are encoded in RNA. [21]

The genetic material has the basic mechanism for translating genetic messages into proteins, which is the main biological molecule type. Proteins have various roles in the cells, and function as, e.g. enzymes or building

blocks. These molecules are the ones that accomplish most of the functions of living cells. Proteins are built out of amino acids, which in turn are built out of nucleobases. An example of a genome is the human genomic material. It contains over three billion base-pairs in the DNA sequence, separated into 23 chromosomes. [21]

2.1.2 Genome features

A genomic feature is a generic term to describe one or multiple regions in the genome with some biological function or feature. A genomic feature can for example be a specific gene, a protein coding sequence, a set of single-nucleobase polymorphisms (SNPs), or the exons for a gene. The regions of genome that make up one feature do not need to have a specific order, but might in some cases. [15]

Genomic coordinate

In order to refer to specific parts of the genome, e.g. a genomic feature, one uses a positional coordinate system called genomic coordinates. Genomic coordinates represents a position or sequence on a specific genome, where a base-pair is located on each position. To represent a genomic coordinate one need to know genome type, chromosome number, strand type and a numbered position, referring the actual position of a base-pair in a sequence of base-pairs, e.g. in a chromosome. There are different conventions for representing genomic coordinates, and one has to define how the base-pairs of a genome are indexed to avoid misinterpretations. The first base-pair in the genome can e.g. start with either the index 0, which is normal in the world of computer science, or 1, which may be more natural for others. Another uncertainty, which has to be defined, is whether a coordinate refers to the actual position of a base-pair, or whether it refers to the gap between two base-pairs, i.e. space-counted. [44]

Genomic region A genomic region is any interval within a genome. To identify such a region, one needs to know the genome, a sequence identifier (e.g. a chromosome) and start and end base-pair position, and in some cases which strand of the genomic sequence one wants.

In this work genomic coordinates are defined to be zero-based and half-opened, meaning that a sequence starts at base-pair position 0, the base-pair positions themself are indexed, and that a genomic region is start-inclusive and end-exclusive. This definition is illustrated in Figure 2.1 on the facing page.

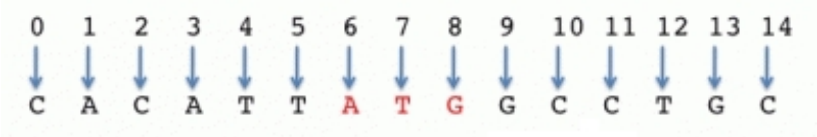


Figure 2.1: The figure shows how genomic coordinates are defined in this work. As seen, it is the base-pairs that are indexed, and the first one is in position 0. The region marked in red has the genomic coordinates [6, 9), as it is start-inclusive and end-exclusive. (Figure from [14])

2.2 Bioinformatics

2.2.1 Genomic tracks

A genomic track is a set of annotations for any genomic feature. Genomic tracks are often called genome annotation tracks or, in a biological setting, just tracks. The features are expressed as data units positioned across a genome sequence. These basic informational data units, or annotations, are called genome elements or track elements, and describe together a genomic feature. Each track element has a genomic coordinate, referring to a position or sequence on the genome, i.e. one or more base-pairs, which can be either a point or a sequence. A genomic track can express either a single or multiple features, and features can consist of one or more track elements. A genomic track may be of a specific type of track, which describes the type of information related to the genome sequence. Genomic tracks are used for biological analysis. Two examples of genomic tracks are a set of track elements that describe the location of genes or the exons of a gene. [20, 35]

2.2.2 Representation of genomic tracks

In order to have a common understanding of a set of genomic track data, one need to have some sort of standardized way of representing it. Many different data formats are used to hold different kinds of genomic data. E.g. methods for representing where SNPs are located in the genome may be very different from how one would represent the temperatures of the melting point for each base-pair spread across the whole genome. These types of data are also typically very different in data space usage, as SNP tracks may only contain a few track elements, but the melting point track would need to be represented with a track element for each base-pair of a genome. To put this in perspective, there are approximately three billions base-pair in the human genome.

In addition, the way humans read genomic tracks efficiently, and how computers read this, may vary a lot. This is something one has to take into consideration when working with genomic data, as the dataset one works with may be very large in size. There are two main formats that are used for representing genomic tracks: *textual formats* and *binary formats*. Each format has their own strengths and weaknesses. Textual formats has, in some cases, a binary version with advantages in different scenarios than the

textual variants. An example of this is the SAM (textual) and BAM (binary) formats for read alignments, where the binary version is faster to read for a computer, but practically unreadable for humans.

Textual formats

A basic representation of a genomic track is a tabular text format only containing positional data, i.e. chromosome identifier and start and end coordinates, for a set of genomic track elements. Each line in the text file represents a track element, and these lines are often sorted on the positional information. The track elements may also include, among other types of information, associated values or relationships between them, i.e. edges in a graph. A tabular text format means that the individual values for each line in the text file are separated by a tab character, i.e. tab-separated values.

Tabular text formats are the most common formats for genomic tracks. This is because they are easily humanly readable, and also simple to parse and manipulate for computers. As mentioned, they consist of tab-separated columns for each data value in the genomic track, making up one track element for each line in the text file. The three most used tabular text formats are Generic Feature Format (GFF), Browser Extensible Data format (BED) and Wiggle Track Format (WIG), each with their own method for representing various types of elements in a track. [20]

BED format The mentioned formats has the ability to represent different, though some overlapping, types of information. The BED format is, for example, used to represent regions in the genome, where each region, or track element, is described by using a sequence identifier and start and end base-pair position (see example in Figure 2.2). These three informational units are in fact mandatory when using the BED format. One can also represent a single point using this type of format by having the end coordinate the same as, or one more than, the start coordinate. The BED format has an additional nine optional fields that can provide more information to the track element if necessary. One of the optional fields is score or value. With this extra field one can, for instance, represent the melting temperatures for every base-pair in the genome.

```
1 #seqid  start   end
2 chr21   1000    2000
3 chr21   2000    3500
4 chr21   3000    4000
5 chrM    200     500
```

Figure 2.2: Example of a tab-separated textual genomic track (a BED file). The hash character (#) denotes a comment.

WIG format While it is possible to represent the melting temperatures for every base-pair in the genome when using the BED format, it provides

the basis for very poor space utilization. This is because every single track element, or line in the file, would have to include the genomic coordinate. An alternative is to use the WIG track format. Here one can specify a region with sequence identifier (e.g. a chromosome), start position and a step value, which describe how many base-pair positions it is between two consecutive track elements. This region is specified as a single line in the text file, and every consecutive line contains only a single data value, being the melting temperature in this example (see example in Figure 2.3). It is also possible to include multiple region lines within one genomic track, e.g. one for each chromosome. [43]

```
1 fixedStep chrom=chr21 start=1 step=1
2 53.4
3 48.1
4 47.6
5 ...
```

Figure 2.3: Example of a WIG file. Lines starting with *fixedStep* denotes a region line.

Binary format

While tabular text files as a representation of genomic track have the advantage of being humanly readable and simple to parse, they are often not very efficient for computers. An alternative representation method of genomic tracks is what is referred to as binary formats. Binary formats are usually more compact and efficient than textual ones, and does often have indexed data for fast lookups. Consequently they are more equipped for high-speed biological analysis. Some of the common textual formats has an associated binary version, e.g. BED has bigBed, WIG has bigWig, SAM has BAM and so forth. [20, 43]

2.2.3 GTrack

Another format for representing genomic tracks is the format called GTrack, which is short for both *Genomic Track* and *Generic Track*. It is developed to serve as a common representation for many of the most used data formats for genomic data. The format is based on four core *informational properties* that provide the basis for fifteen different *track types*. These two concepts are further described in the following paragraphs. GTrack is a generic, space optimized and type-aware tabular text format. It is generic in the sense that it can represent a union of many other normal track formats, and the mentioned fifteen track types. Space optimized because each track type is represented with a minimum amount of information, i.e. flexible number of columns. And type-aware because metadata such as track type information, format-specific characteristics and tool-specific format requirements are all described in the file header. The format emphasizes preciseness, flexibility

and simple parsing. It also incorporates independent data lines. This means that each line in the file can be interpret independently of its location, although there are some exceptions. [20]

Informational properties

The GTrack format is built on four core informational properties that can be used to describe genomic tracks. These four properties are *gaps*, *lengths*, *values* and *interconnection*. The positional information, i.e. start and end base-pair, of track elements in a genomic track are described as the *lengths* of the track elements and the *gaps* between them. Each track element may also have an associated *value*. This can be any type of value, e.g. a number, character or boolean value. The value property may also be vectors of values, however with the restriction that each element in the vector has the same datatype. Lastly, track elements may be connected to one another with edges, or *interconnections*. This is necessary for describing three-dimensional data. Interconnections can also have an attached weight.

GTrack has four core reserved columns, *start*, *end*, *value* and *edges*. These corresponds to the four core informational properties, gaps, lengths, values and interconnections, respectively. Each combination of these properties uniquely give a certain track type. There are a total of fifteen combinations, and hence fifteen track types. This is because each property can either be present or not, and with four properties, this gives $2^4 = 16$ different possible combinations. We ignore the one combination where none of the four informational properties are present, as it would only produce an empty track. This results in the fifteen different track types. [20]

Track types

The type of a genomic track is defined by what information it contains. In Figure 2.4 on the next page the different track types are illustrated. The figure shows which track type that is derived from the different combinations of the four core informational properties. It is illustrated using a four-dimensional matrix where each axis is one of the core informational properties. The top-left cell in the top-left box in Figure 2.4 is not considered a type of track as it contains no information. [20]

Sparse and dense tracks The Figure 2.4 shows all the different track types that are derived from the informational properties. The track types can be divided into two groups: *Sparse* and *dense*. The sparse tracks are the ones that include *gaps*, while the dense are the ones that do not include any gaps. In the GTrack format, this means that tracks that have the attribute *start* are sparse, while the ones without *starts* are dense. If a track includes neither *starts* nor *ends*, the track has a track element for every base-pair position within all the bounding regions of a track. This is sometimes referred to as that the *representation* is dense. Function tracks are examples of such tracks.

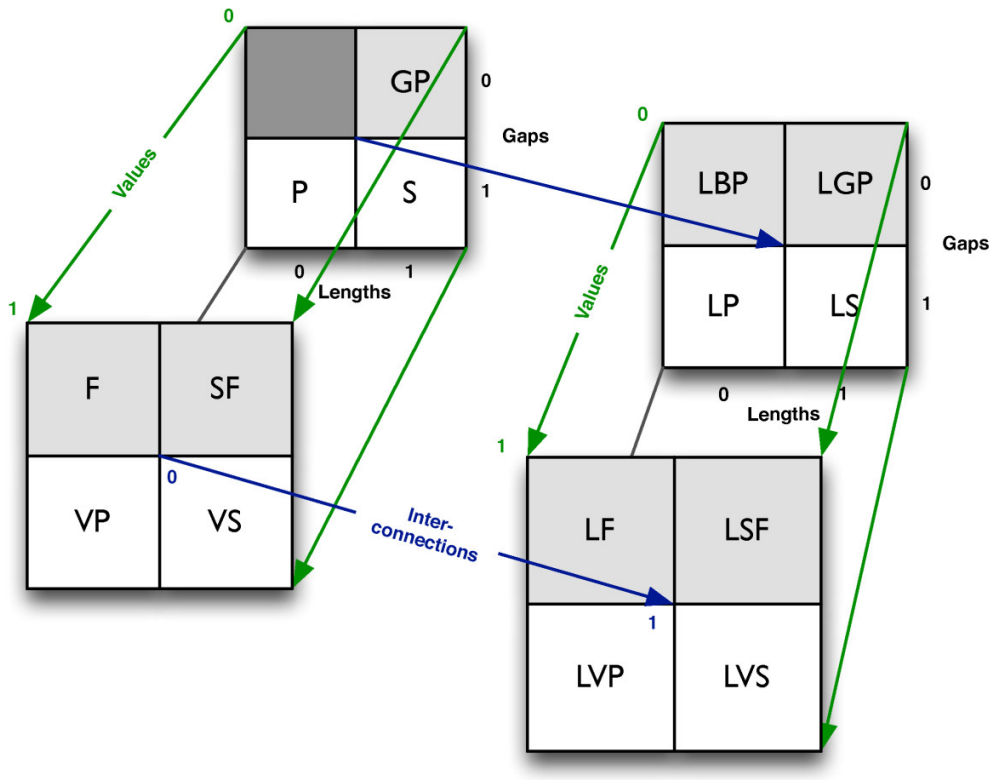


Figure 2.4: **Illustration of what information that is associated with each track type.** Abbreviations: Genome Partition (GP), Points (P), Segments (S), Function (F), Step Function (SF), Valued Points (VP), Valued Segments (VS), Linked Base Pairs (LBP), Linked Genome Partition (LGP), Linked Points (LP), Linked Segments (LS), Linked Function (LF), Linked Step Function (LSF), Linked Valued Points (LVP) and Linked Valued Segments (LVS). The boxes with grey background represent dense tracks, while the ones with white background are considered sparse. (Figure from [20])

The reserved attributes In addition to the four core informational properties that are reserved column names in the GTrack format, there are four more reserved columns. These are *genome*, *seqid* (sequence identifier), *strand*, and *id*. The id column is a required column if the track is used to describe interconnection, or edges, as it is used to uniquely identify the elements of a track.

Bounding regions

A bounding region describe a genomic interval in a genome that may contain genome elements. The bounding regions of a track is the regions for where the track contains information, i.e. the *domain* of the track. No genome element can be located outside any bounding region, or cross any of their boundaries. Bounding regions can be whole chromosomes or any other smaller region. Additionally, no bounding region is allowed to overlap

another bounding region.

It is worth noting that for sparse tracks, the base-pair positions that are not covered by any genome elements, i.e. the absent of any genome elements, are also considered informational parts of the track. For dense tracks, all the genome elements, by definition, entirely fill the domain of the track, i.e. all base-pair positions are covered by a genome element.

Figure 2.5 shows a high-level model of the structure of genomic tracks, and how bounding regions fits into the track model.

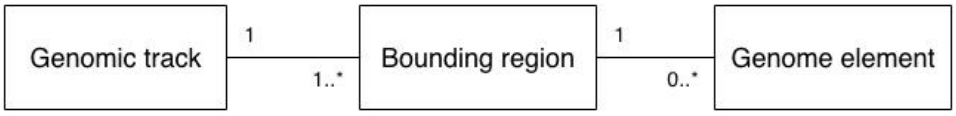


Figure 2.5: **An abstract illustration of a genomic track.** At the highest level of abstraction, a *genomic track* is defined to be a genomic sequence that contain a set of one or more non-overlapping *bounding regions*, that in turn contain zero or more arbitrarily informational units as *genome elements*. While the term *bounding region* only is used in the setting of the GTrack format, it is also an implicit concept in all the other formats that represents genomic tracks.

2.2.4 Analyzing genomic data

In order to do research based on the genomic track data, one has to conduct some kind of analysis. There are various tools and applications for doing so, such as BEDTools [6] for analysis on home computers, and Galaxy [19], which is a comprehensive system for biological research in the cloud[1]. Analyzing genomic data very often involves performing some sort of operation to the data set. This can be anything from counting number of track elements in a track to merging two tracks into one. These example operations result in an integer and a new track, respectively.

Track operations

For analysis of genomic tracks, various statistical investigations are performed to generate results that have a biological significance. Depending on the type track in an analysis, and involvement of multiple tracks, different statistical investigations, or operations, can be performed, giving different types of data for different analytical purposes.

Without considering the scientific purpose, a few examples of basic operations one can perform on a single genomic track is count of points for Points tracks, base-pair coverage within Segments or a mean value for a Function track. When introducing another track one can generate data like number of points in track 1 that are within segments in track 2 or mean value in a function in track 1 for the points in track 2. Moreover, one can,

[1] Galaxy is also available for home computers. [16]

for instance, perform operations that investigate whether the segments in two Segments tracks are relatively close to one another or far apart.

Track data are represented in specific formats so that analysis, such as the above mentioned, can be performed efficiently. An example of this is a search for elements in specific areas on the genome. A basic representation of a genomic track is a tabular text file only containing sorted positional data for each track element. For a format like this a search for elements in specific areas are trivial as it is only a simple traversal through the track elements stopping when the search has passed the particular area.

2.2.5 The Genomic HyperBrowser

The Genomic HyperBrowser is a comprehensive system for analyzing genomic track data and to provide statistical analysis on said tracks as either descriptive statistics or hypothesis test. It is an open-ended system, which means that it provide a core set of abstractions, rather than having a few usage scenarios. The HyperBrowser supports the most common textual formats, with the addition of the GTrack format, which distinguishes between the fifteen different types of tracks (described in the section 2.2.2). Different track types for two genomic tracks provide multiple possible analyses to be performed. E.g. for a track with points and a Function track it is possible to statistically investigate whether there are higher values at the points than elsewhere. The HyperBrowser is tightly integrated with the Galaxy framework with regard to its tools, web access and sharing possibilities. [35, 36]

Galaxy

Galaxy is an open, web-based platform for biomedical and genomic research that automatically tracks and manages data. Users can perform computational analysis of genomic data and use various tools, such as data visualization, workflow charts and data sharing. Galaxy focuses on three main aspects; accessibility, transparency and reproducibility. The users data, e.g. histories, pages, data sets etc., in Galaxy can easily be publicly shared and accessed by other Galaxy users for their own reproduction and reuse. [19] Galaxy supports most of the major file formats used for analyses, such as BED, GFF or BAM for sequence alignment. [13] It includes a large set of tools for performing operations on, for instance, interval tracks, or Segments tracks, such as intersect, subtract, coverage and join. Additionally, Galaxy allows user to create their own tools. [22]

2.2.6 GTrackCore

The GTrackCore is a module of the HyperBrowser that handles track preprocessing and track data retrieval. As of now it is integrated in the HyperBrowser code base, but can to a certain extent be considered as a stand-alone module. This is because it is somewhat loosely coupled to the main part of the code base, i.e. the statistical analysis algorithms. There are only a few exceptions to this, such as track and genome metadata,

and common functions. Future plans of the GTrackCore is to make it as a completely stand-alone and flexible module that can be used in other settings than in the HyperBrowser. One could for instance create a set of command line tools, which only would need the GTrackCore, and not the entire HyperBrowser, to perform statistical analyzes on genomic track data, equal to BEDTools [6].

Track preprocessing and storage

Track preprocessing is the process of converting raw textual track data into a binary format, and storing it in a suitable way on disk. This data can be accessed later in a much faster manner than normal textual, unprocessed track data.

GTrackCore understands, and can preprocess, many of the well-known textual formats, and is not restricted to only the GTrack format.

Track data retrieval

Data retrieval is the task of retrieving the binary data, which was created by the preprocessor. The track data is extracted from the GTrackCore using an interface called a *track view*. This is, as the name suggests, a view over a genomic track, with capabilities for retrieving data in a fast and efficient way. From this view one can retrieve data in two ways; either iterate through all the track elements within the view, or by retrieving the elements as a NumPy array (one NumPy array for each of the columns of the genomic track).

2.3 Technology

2.3.1 Python

Python is an interpreted and object-oriented programming language. It is a suitable tool for computational means because of the large amount of free and open-source libraries for effective scientific computing, such as SciPy [37], and specialized libraries for biological computations, such as Biopython [7]. The programming language is a very high-level language and can also be used as a scripting language. It might be easier to learn than many other programming languages because of its clear and simple syntax. The standard Python containers is considered to be high-level structures, which include e.g. lists, dictionaries and tuples, and are not suited for high-performance numerical computation. [32, 46]

2.3.2 NumPy

NumPy (Numerical Python) is a fundamental Python library for scientific computing. It provides, among other features, an alternative to some of the standard Python containers. NumPy uses n-dimensional arrays and matrices, called *ndarray* objects, or *NumPy arrays*, instead of multidimensional Python lists, and includes an assortment of fast operations

for these. This includes operations such as mathematical, sorting, selecting, I/O, statistical and many more for calculation and manipulation. NumPy is partly written in C for efficiency purposes, and is considered to be the standard way of representing numerical data in Python. [27, 46]

NumPy vs. standard Python containers

The NumPy ndarray objects are considerably faster for performing calculations than the standard built-in Python containers. This is because of the underlying structure of the type. Lists in Python grow dynamically when elements are inserted or removed, as opposed to ndarrays that have a fixed size upon creation. Elements in ndarrays are also obliged to be of the same data type, unlike Python lists where numbers, strings, literals, and even lists, can be elements of the list. Because ndarrays only consist of a single data type and have a fixed size, it can be stored in contiguous memory. As a result of this it can be accessed more efficiently when performing operations than the standard Python containers, where each element has to be fetched from memory and also dynamically type checked, as they under the hood are represented as a fixed array of pointers to any Python object. [27, 46]

2.3.3 Memmap

Memmap, which is short for memory-map, is a subclass of the NumPy ndarray and is a type of object that provide the opportunity to store ndarray objects as, possibly large, files to disk as binary data. One can later access and read segments of the data quickly, without necessarily reading the entire file into memory. This is useful in situations where files are very large (in the order of gigabytes), as genomic data often may be. [24]

2.3.4 HDF5

HDF5 (Hierarchical Data Format, version 5) is a data file format designed to store and organize large amounts and complex data collections on disk. Complex data is a composition of basic data units (integers, floats, etc.) and other complex data, meaning that practically any type of data can be represented. The data is organized in a hierarchical, UNIX filesystem-like [45] format, with *groups* (HDF5 Group), resembling directories, and *datasets* (HDF5 Dataset), resembling files. These are the two basic structures of HDF5, and are referred to as HDF5 objects. A group can contain zero or more groups or datasets. The datasets are multidimensional arrays of a homogeneous data type. The HDF5 technology suite includes an official high-level API for a range of programming languages, including Fortran, Java and C. HDF5 also support compression of datasets. [40, 41]

Abstract data model

File The HDF5 file is the container which holds all the other HDF5 objects, i.e. the groups and datasets. The objects within a file are uniquely identified by their *name*, and are organized in a hierarchical structure.

The objects are either members or descendants of the *root group*: A group which is required in all HDF5 files.

Group The group object is a structure that parents zero or more other HDF5 objects. A *group header* contains its name and table of children.

Dataset The HDF5 dataset object is a multidimensional array of a homogeneous data type. A dataset is comprised of a header and a data array. The header includes the following essential information:

Name The unique identifier of the object.

Datatype A description of the type of data of the dataset. There are two datatype categories, *atomic* and *compound*:

An atomic datatype is any of the basic data types, such as integers and floats.

A compound datatype is a collection of data types represented as a single unit, and is very similar to a *struct* in C [23, p. 150-179]. The members of a compound datatype can be both atomic or compound datatypes.

Dataspace The dataspace describes the dimensions of a dataset, and includes the properties *rank*, i.e. the number of dimensions, and *size of the dimensions*.

Storage layout The HDF5 format supports two storage layouts. The default is *contiguous*, i.e. linearly. The other is called *chunked*, which is further described in the next section.

Chunked storage layout

The chunked storage layout is a method where datasets are divided into blocks of equal size, called *chunks*, which are stored separately on disk, i.e. not contiguously. The HDF5 library treats chunks as atomic objects, and disk I/O is always in terms of complete chunks. This storage method makes it possible to achieve good performance when accessing subsets of a dataset, and allows for I/O *filters*, such as compression and encryption, on entire chunks of data. These filters are applied on a complete chunk for every disk I/O. For this to perform well one has to specify the shape and size of the chunks of a dataset according to its usage scenarios. [10]

Too small chunks The smaller the chunks, the more of them in a dataset. If a selection of a dataset cover many chunks, more disk I/O operations needs to be performed than if it cover few chunks. There is also metadata associated with each chunk, making the file size larger if there are many chunks.

Too large chunks Since HDF5 always performs disk I/O in terms of complete chunks, and applies any potential filters on them each time, selections of subsets, especially small subsets, of a dataset may affect performance poorly.

Caching chunks The HDF5 library makes sure to cache whole data chunks when they are read to possibly decrease disk I/O. This is done after any filters have been applied to the chunk.

2.3.5 PyTables

PyTables is a package for Python that is designed to efficiently work with large datasets. The underlying foundation of data organization is built on top of the HDF5 format. PyTables takes advantage of NumPy for in-memory handling of the large datasets, and uses Cython, a C extension for Python [12], for performance-critical code parts, which ensures fast operations. While HDF5 only stores homogeneous arrays, the PyTables package, built of top of the data format, includes heterogeneous *table* structures with querying and indexing capabilities.

A feature of PyTables is that it optimizes memory and disk resources, and hence uses less space than other solutions, such as relational databases, especially with compression turned on. The main point about PyTables is that all the datasets are resident on disk until you explicitly read it, or more commonly, read parts of it. A PyTables file stored on disk is a HDF5 file, with extra metadata and structures to the objects within the file. [1]

The object tree

Because PyTables is built on top of HDF5, it too implements a hierarchical approach to its inner data structure. The data is organized hierarchically in *groups* (`Group` class) and *leaves* (`Leaf` class). All objects in the tree are descendants of the `Node` class, and are generically referred to as *nodes*. *Groups* have zero or more children, which can either be other groups or leaves. The *leaves* are the actual dataset objects. These can either be homogeneous or heterogeneous sets, which in PyTables is represented by either *arrays* or *tables*, respectively.

All datasets in PyTables are stored in the HDF5 format using the chunked storage layout (described in section 2.3.4 on page 13), with the exception of the homogeneous dataset type `Array` (but not its subtypes).

File object The `tables.File` objects is an in-memory representation of the PyTables file. An object of this type is returned when calling the PyTables `open_file` method, and includes methods for manipulating the objects of a PyTables file, such as creating and removing nodes. The `root` attribute of the File object is the entry point to the rest of the nodes in the tree structure.

Homogeneous datasets Multidimensional homogeneous datasets in PyTables are stored as *arrays*, and is represented by the `Array` class or either of its sub-classes `CArray`, `EArray` and `VLArray`. The arrays of PyTables are generally used in the same fashion as NumPy arrays, regarding reading and writing. Homogeneous datasets are stored in the HDF5 format with an

atmoic datatype. Below is a list of the available array types of PyTables, together with a summary of their properties.

Array A quick-and-dirty array storage with fast I/O, which should only be used with relatively small datasets, i.e. those that fit in memory. An extra feature with the regular array type is that it remember its flavor. If one, for instance, persists a Python list container, one will also retrieve a Python list. A requirement is that the container is not nested (e.g. not lists within lists) and that all the elements are of the same datatype.

CArray The difference between compressible arrays and normal arrays is that the former uses a chunked storage layout, which is a premise for compression and provides efficient performance when accessing subsets of a dataset. Both writing to and reading from disk is fast for this array type.

EArray An EArray is equal to a CArray, with the exception that it also is *enlargeable*. The writing speed is approximately as fast as with the CArray, while reading is equally fast. For EArrays one have to specify a shape where one, and only one, dimension is set to 0, which is the enlargeable dimension of the dataset (see Figure 2.6 on page 19). While the HDF5 format support datasets to grow in multiple dimensions, PyTables is restricted to only one.

VLArray The VLArray (variable length array) supports collections of homogeneous data with a variable number of entries. Equal to the EArray, it is both compressible and enlargeable, but disk I/O is not very fast.

Heterogeneous datasets To persist heterogeneous data with PyTables one uses *tables*, which is represented by the `Table` class. A table is a sequence of *rows*, each with one or more uniquely named *fields*. All the rows of a table has the same fields. The fields are arranged as *columns*, whereas each is a homogeneous multidimensional dataset, with the exception of fields with a nested structure, similar to complex datatypes. Heterogeneous datasets are stored in the HDF5 format with a compound datatype.

An example of how to create a table in PyTables is seen in Figure 2.6 on page 19 from line 11 to 13. The table is created at the root node (`'/'`) in the PyTables file (`h5`), with the name `person_table` and the columns that are described in `table_description`.

The use of tables in PyTables is not a replacement to e.g. a relational database, mainly because relations across tables are not possible, other than the hierarchically relationship of the datasets. The PyTables library do, however, provide some typical database functionality for tables, such as *querying* tables and *indexing* columns. In addition to retrieving data from tables by queries, they also support retrieval from columns by slicing the `Column` object (line 21 in Figure 2.7 on page 19), i.e. implicitly calling the special Python method `__getitem__(key)`. Conversely, one can assign

multiple values to a column with the method `Column.__setitem__(key, value)`. It is also possible to iterate the `Row` objects in the table sequentially, which can be further specified by providing start and stop indices in the table (line 10 in Figure 2.7 on page 19). Here, one have access to all the fields of the row, with the ability to update them if the PyTables file was opened in write mode.

Queries PyTables include in-kernel searches, using a query engine that efficiently can retrieve rows from table that fulfill a given criteria. To query a table, the *where* methods of the `Table` class is called, and is passed a string with the required conditions for the search. The string must involve at least one of the columns of the table, and optionally constants and variables, all combined with algebraic operators and functions. These condition strings are interpreted by Numexpr, which is a Python package for achieving fast computation of array operations. For arrays that are larger than the CPU cache, Numexpr can even be faster than NumPy because it avoids creating temporaries for intermediate results. [26] With complex conditions, spanning multiple columns, PyTables support simultaneous searching with the in-kernel engine. The *where* statements returns an efficient iterator that yields the rows that fulfills the requested criteria. An example of a PyTables query is seen on line 17 in Figure 2.7 on page 19.

Indexing To speed up the query engine, PyTables include column indexing for non-nested columns, i.e. with an atomic datatype. The search methods automatically takes advantage of indexed columns. PyTables uses OPSI (Optimized Partially Sorted Indexes) to index table data, which is a indexing engine that allows PyTables to perform fast queries on very large tables. OPSI is optimized to be used with read-only or append-only data. Updates and deletions are, however, much slower than other solutions. [28]

The query on line 17 in Figure 2.7 on page 19 would automatically use an index if it exists. To create an index for a table column, one calls the `create_index` method of the `Column` object:

```
person_table.cols.age.create_index()
```

Note that the PyTables file has to be opened in write mode to create column indices.

Compression

To reduce the amount of time spent on disk I/O, especially for large datasets, and to decrease file size, PyTables includes compression of datasets. PyTables support a small variety of compression methods, such as zlib [48] (the standard in HDF5) and Blosc [8], each called a *filter*. The latter was developed to cover the needs of PyTables, and is a high-performance compressor optimized for binary data. The level of compression is customizable by the user. When compressing datasets with a low compression level, the lower disk read time can compensate

for the decompression time, making the total read time lower than with uncompressed data.

Automatic chunkshape

If one provides an estimate of how many elements a dataset will contain, PyTables will automatically set a sensible chunkshape of the HDF5 dataset, and try to make it as efficiently as possible for both insertion and retrieval. Automatic chunkshape in PyTables is optimized to make I/O perform well, but not best, for all usage scenarios. If one knows that an application always will do data selections that are very small or very large, it is possible to specify the shape explicitly to optimize I/O. [29] An example of automatic and manual chunkshape is seen in Appendix D.

```
import tables

h5 = tables.open_file('data.h5', mode='w')

h5.create_group('/', 'person_arrays')
h5.create_earray('/person_arrays', 'name',
                 dtype=tables.StringAtom(50), shape=(0,))
h5.create_earray('/person_arrays', 'age',
                 dtype=tables.Int8Atom(), shape=(0,))

table_description = {'name': tables.StringCol(50),
                     'age': tables.Int8Col()}
h5.create_table('/', 'person_table', table_description)

h5.close()
```

Figure 2.6: Example of how to create *groups*, *arrays* (`tables.EArray`) and *tables* in PyTables. For arrays, one specifies an atomic datatype, while the columns of a table is described by `Col` instances, which are directly derived from the `Atom` classes.

```
h5 = tables.open_file('data.h5', mode='r')

# get table and array object using natrual naming
person_table = h5.root.person_table
name_array = h5.root.person_arrays.name
# get array object using the get_node method
age_array = h5.get_node('/person_arrays/age')

# iterate the rows of a table from index 100 to 200
for row in person_table.iterrows(start=100, stop=200):
    print row['name'], row['age']

# retrieve a list of row tuples from index 100 to 200
print person_table[100:200]

# query the table for all persons with age > 25
for row in person_table.where('age > 25'):
    print row['name'], row['age']

# retrieve NumPy arrays of the whole age column and age array
print person_table.cols.age[:]
print age_array[:]

h5.close()
```

Figure 2.7: Examples of data retrieval in PyTables. It is assumed that PyTables is imported and that values are put in the datasets.

Chapter 3

The development process and practices

This chapter gives an overview of the development process and some insight to the development tools we used.

3.1 The development process

3.1.1 Understanding GTrackCore and PyTables

In the early phases of the development process, most of our effort was concentrated on getting acquainted with and understanding the already exiting code base of GTrackCore. In order to know how to replace the storing and retrieval mechanism, and what to replace, we had to learn how this was already done in the old GTrackCore version. As the code base is fairly large ($\sim$ 21000 lines of code), this was a partially demanding task.

To obtain deeper insight to the exiting GTrackCore we took advantage of *assumptive documentation*, which is a method for learning others code. [3] Our approach was documenting various functions, classes and modules by making assumptions of how they worked. These assumptions were later confirmed or corrected together with the original code writers. When later having acquired a basic understanding of the code base, we aimed for developing the new GTrackCore.

Moreover, creating an UML diagram of many of the main parts of GTrackCore was a second method applied for deeper understanding later in the development process. This helped reveal dependencies of many of the different parts of GTrackCore, and how these interacted with each other.

Throughout the process we also contacted the developers of PyTables on various occasions, seeking information that was not available in the documentation. [1]

[1] https://github.com/PyTables/PyTables/issues/338, https://groups.google.com/forum/#\protect\kern-.1667em\relaxtopic/pytables-users/hA5OMF8WY18, https://groups.google.com/forum/#\protect\kern-.1667em\relaxtopic/pytables-users/bfNGat30S7o

3.1.2 Developing the new GTrackCore with PyTables

We started the development by creating a minimal preprocessor prototype that could process a simple Segments track with a fixed set of attributes, and store it in a PyTables file, using tables as dataset type. This step was more of a proof-of-concept, where we tested PyTables functionality and how it matched up with needed functionality of GTrackCore. An important functionality was the ability to store data in a way that made it possible to retrieve it as NumPy ndarray objects, using the same datatypes as the one used in the previous GTrackCore version.

To investigate whether the new PyTables solution was an at all viable alternative to the old memmap solution, we did some minor small-scale performance tests. This mostly included testing how fast we could retrieve data from a single column of tables. When these tests showed results that was within the same order of magnitude as with memmaps, we started evolving the prototype to a general preprocessor that could process any type of track that the old version could do. Since the preprocessor already was thoroughly tested by the developers of the old GTrackCore, we employed the same tests to ensure that the new version supported the exact functionality as the old. A more detailed explanation of how we used the tests is described in the next section.

From the retrieval part of GTrackCore, we created new versions of central classes in the existing code base, such as the classes named `TrackSource` and `TrackViewLoader`. Moreover, we tuned the existing data interfaces, i.e. the `TrackView` and the `GraphView`, to work properly with PyTables. This mostly included utilizing the Row iterator of the `Table` class for iteration. All the details regarding the implementation of the data retrieval part of GTrackCore is presented in Chapter 5 on page 31.

To sum up, the main development phase consisted primarily of adding the needed functionality that made the predefined *tests* pass. Additionally, we continually *refactored* and *optimized* the solution as we understood GTrackCore better, which we will come back to later.

Testing

The existing GTrackCore already included a large set of unit tests and integration tests that covered most of the functionality of GTrackCore. These were implemented using the Python testing framework *nose* [25].

In our case, the most important tests were those that preprocessed all the different track types with various parameters (called *testAll*), and then retrieved the persisted data to assert its equality to the input data. The tests proved to be a very important tool in our development process, as they revealed many of the deficiencies and weaknesses in our solution. There were, however, a few cases of functionality of GTrackCore that the tests did not cover. For instance, a case when preprocessing a track where the original textual data is split into multiple files that caused a bit of problems for us during the development, known as the *shape issue*, which is described in detail in [34].

Refactoring

Refactoring is the process of restructuring existing code, without changing its external behavior. This is usually done to improve code readability, reduce complexity and possibly make the code more general and flexible.

At the early stages of the development process, we had little deep knowledge of Python, PyTables and HDF5, and in particular GTrackCore itself. Hence, our initial code design and style was at best mediocre. Consequently, our code has been subject to a certain degree of refactoring as our understanding of the mentioned subjects gradually increased throughout the development. Most of these are very small and seemingly insignificant changes, but have to some extent increased the readability of the code. Furthermore, we did not have much experience with choosing correct design patterns for a given setting, and thus did not always choose an appropriate design for the parts we replaced early on. These were later changed as we gained a more comprehensive picture of the GTrackCore system and details surrounding the PyTables library and the HDF5 file format.

The refactoring was a continuous process that went on throughout the whole project as an iterative and incremental process. This type of process is part of the software development process known as *agile* [4, 5].

Details regarding some of the larger refactors, structure changes and also optimizations we made throughout the development will be presented in Chapter 6 on page 55 to shed light on the agile development style.

3.2 Development practices

This section introduces the chosen approach to the developed code, i.e. the development practices. This includes how we performed the collaboration, using pair programming, and methods to organize and keeping track of changes throughout the process.

3.2.1 Collaborative work

As mentioned, most of the practically work for this thesis, i.e. implementing the new GTrackCore, is done as a collaborative work process. Collaboration generally improves the problem-solving process by having more people working towards a common goal, continuously exchanging knowledge through discussion. [47]

Pair programming

Pair programming is the process of having two people, side by side by the computer, together producing program code as equal partners. Person A – the driver – writes the code, while person B actively observes the driver, looking out for defects, considering and proposing alternative methods for the specific problem at hand. The roles are regularly switched. This has proven to be a more efficient method than having the two programmers working on separate tasks. When using pair programming, code quality

is increased, the design is better and the programmers generally feel more satisfied with the end result. [47]

All of the design choices, and most of the produced code, was made collaboratively. The only exceptions to this was many of the easier tasks, such as writing tests and some minor refactoring jobs that increased the code readability.

3.2.2 Version control

When working in larger projects together with multiple participants, it is most important to keep track of, and log, all the changes in the progress in case something goes wrong, or one wants to revert the project to a previous state. For this purpose we used a version control system, *git* [17] and *GitHub* [18], which, with a simple interface, helps developers take note of changes, upload them to web, and making them available for other developers with access to the project repository.

3.2.3 Code style

The source code of the old version of GTrackCore is written using `camelCase` [9], while we have used `snake_case` [39], following the style guide for Python code [30]. We have done this for two reasons: firstly to follow coding conventions, and secondly to purposely separate code from the old GTrackCore in the new one to easier see changes.

Pythonic code

As our general Python competence increased throughout the development process, we incorporated a more *Pythonic* approach to our code. This includes, among many other styles and idioms, code that is simple and readable with a flat design, rather than nested. [33] Some of the refactoring we did was a result of wanting to achieve a more Pythonic code base, so that later readers of the code easier can understand it.

Documentation

Writing documentation for problems that are not perfectly understood from the start is both difficult and time consuming, especially in a research setting. Additionally, the code is continually subject to refactoring, which leads to that the documentation has to be rewritten accordingly. As a result, we focused our documentation on the following:

- Self-documenting code, i.e. explanatory method and variable names
- Explanatory messages in the version control log, with frequent and enclosed changes

Due to our short time-frame to complete such a comprehensive implementation, we saw it fit to prioritize "working software over comprehensive documentation" – [5].

Chapter 4

Methods of measurement

In this chapter we present the methods that were used to compare performance of the old and new GTrackCore versions and what technique we used to optimize the new solution.

4.1 Overview

The main focus of this work was to re-implement large parts of GTrackCore to operate with a new data model, i.e. the PyTables library that makes use of the HDF5 format, and to investigate whether PyTables is a suitable replacement to the existing memmap solution. To support an answer to this question, one has to consider different results, one of which is the source code itself, i.e. the code quality of the implementation. Another factor is the performance of the implementation, compared to the old version. These results will be presented in Chapter 6 on page 55 and Chapter 7 on page 63, and will consist of the following:

1. Code improvements made during the development that lead up to the final result, which includes:
    - Refactoring that has improved code quality
    - Optimizations that has improved performance
2. Comparison of the old and new GTrackCore
    - Comparisons of the PyTables version with different settings

4.2 Code measurement

While code quality is difficult to characterize and measure, we will, in this work, look at the following factors:

- *Readability*
  Readability relates to whether a class, a module or a method is easily readable, concerning the need for others to understand it later. This includes code that is simple, explicit and uncomplicated, with explanatory variable names.

- *Maintainability*
  The code base is maintainable if it easily can be extended or altered. A typical trait of maintainability is code parts separated independent and multipurpose modules.

- *Efficiency*
  While efficiency is not a measurement of code quality itself, it is used to measure performance using execution speed as metric.

It is important to note that the first two are not meant for comparison between the the old and new version of GTrackCore, but rather a way of describing why certain choices were made regarding the implementation. These terms are only used in Chapter 6, where major code improvements are presented and analyzed.

4.3 Profiling

To discover bottlenecks in a program, one can use a *profiling tool* to investigate it. Profiling is a form of analysis tool that is used to measure programs, primarily to aid in optimizations. A typical analysis is to measure the duration of function calls to discover which functions that may be the bottlenecks of a program. To profile and analyze our implementation, we have used a Python module called *cProfile* [11].

The type of profiling used is called *deterministic profiling*, which monitors all function calls, function returns, and exceptions, and measures time between these events, i.e. when the actual program code is running. While there is a overhead in this type of measurement, it is very small due to the fact that there is an interpreter present in Python during execution providing hooks for each event.

The profiles output time as elapsed wall-clock time, i.e. the human perception of time, like a clock on the wall, and not in CPU time of the process. CPU time measuring does not include disk I/O, which is important in this work as we measure operations that deal with heavy disk reads. A limitation of deterministic profiling is that the accuracy is only as good as the clock frequency of the CPU, typically at around 0.001 seconds. [42]

Profile output

When performing a profiling of an execution through the HyperBrowser tools we created (see section 4.5 on the facing page), a detailed list of each function call is presented, which shows total duration and the number of calls to each function. In addition, a dump of the profile data is downloadable. This can be used by other programs that are designed for analyzing profiles, such as *RunSnakeRun*[1].

The following is an example of a typical profile output, with an explanation of the output underneath:

[1] http://www.vrplumber.com/programming/runsnakerun/

```
62576 function calls (61427 primitive calls) in 3.569 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   3/1    0.000    0.000    3.569    3.569 :1()
     1    0.025    0.025    3.569    3.569 <fn>:145(overlap)
    24    1.243    0.052    2.860    0.119 <fn>:117(overlap_track_views)
...
```

function calls Total number of function calls, including recursive calls

primitive calls Total number of function calls, excluding recursive calls

ncalls Number of calls to a function in the form `<recursive>/<primitive>`

tottime Total time spent in a function, excluding time spent in any sub-calls

cumtime Total time spent in a function, including time spent in all sub-calls

percall Average time spent for each function, one for each of tottime and cumtime

filename:lineno(function) Filename, line number and function name

4.4 Benchmarking

A benchmark is a standard by which something can be measured or judged, and has different meanings and units of measurement for various domains.

In the setting of this work, benchmarks are used to measure the execution time of an operation performed by GTrackCore by calculating the running time spent using the *average of N runs*. The benchmark results are used to compare the old and new versions of GTrackCore to help indicate whether the new version is a viable and suitable future replacement.

The results from the benchmark testing will be presented in Chapter 7 on page 63.

4.5 HyperBrowser tools for testing

To systematically compare the old and new GTrackCore, we created two HyperBrowser tools for benchmark testing: *Preprocessor performance tool (storage)* and *Operation performance tool (retrieval)*, each created for testing the performance of the preprocessor and data retrieval, respectively. The tools are available from a separate development instance of HyperBrowser, found in Appendix B.

The reason for using the HyperBrowser platform to perform the benchmark tests and comparisons is that it provides functionality for

displaying results in an organized and transparent manner. Each result of a program execution is accessible from a history, which easily can be distributed and reproduced. Additionally, the tools would be executed in a realistic setting, i.e. on the same server, and on the same hardware, as the production instance of HyperBrowser, called *insilico*, which is hosted at the University of Oslo.

4.5.1 Operation performance tool

The operation tool has two separate functionalities: *Detailed profiling* and *Average of N runs*, i.e. benchmark testing. The first shows a detailed profiling of one operation execution, which displays a list of each method that was called, along with number of calls for each method and the duration of each call. The second lets the user specify the number of times to perform a given operation, and then finds the average of time spent for each execution. In addition to choosing which type of test to run, one also has to specify which GTrackCore version one want to use and what operation to perform.

The set of different track operations that the tool support are the ones described in section 5.10 on page 53. To test various aspect of GTrackCore, it is possible to specify a small range of options that affect the outcome of an operation. This includes specifying how many track views one wants to use, and for the PyTables version, what compression level that should be used, or whether to read from the table or the arrays. These variances will be described in detail in section 7 on page 63. Two examples of the operation tool is seen in Figure 4.1 on the facing page.

We also make sure to remove any page cache[1] between test runs to ensure more reliable results. This is because a lot of the operations used in testing perform disk read operations. Hence, when performing 'Average of N runs' testing, page caching will not affect the result.

4.5.2 Testing environment

The *insilico* server – where the HyperBrowser instance is hosted – is a Linux server, with 32 physical CPU cores and about 500 GB of available memory. Moreover, the following software versions were used in testing:

- Python version: 2.7.6
- NumPy version: 1.7.1
- PyTables version: 3.1.0
- HDF5 version: 1.8.9

[1] http://en.wikipedia.org/wiki/Page_cache

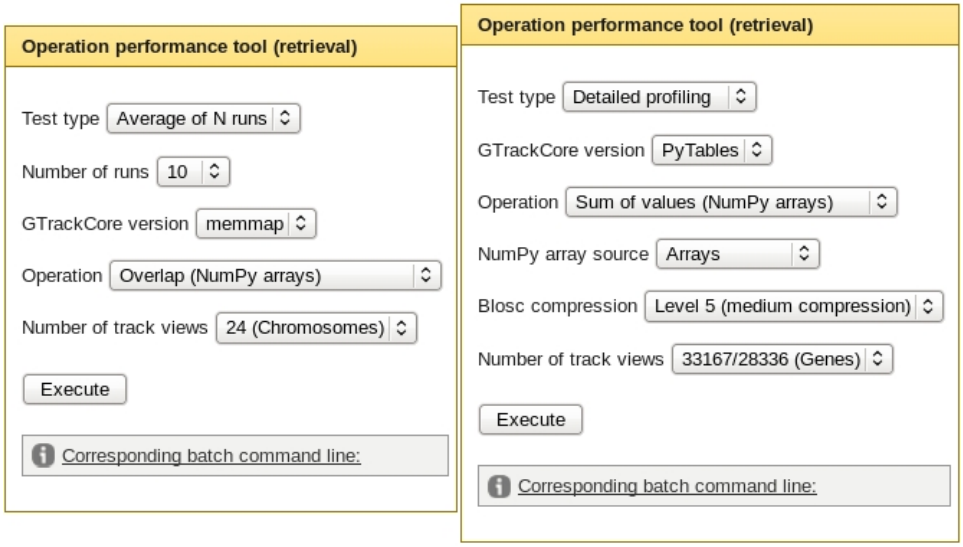


Figure 4.1: The figure shows two screenshots of the *Operation performance tool*. At the left, the memmap version of GTrackCore is chosen to perform a benchmark test for the operation *Overlap (NumPy array)*. The next screenshot shows a setup for profiling the PyTables version for the operation *Sum of values*.

Chapter 5

Implementation of PyTables-based data retrieval in GTrackCore

This chapter presents the implementation details regarding the new version of GTrackCore and some specifics from the old one, mostly focusing on the data retrieval part of GTrackCore. The introducing section provides a short explanation of how the two GTrackCore versions are referred to.

The full implementation is available in our GitHub repository in Appendix A.

5.1 Overview

One can simplify the difference between the two GTrackCore versions by saying that the first one is memmap-based, and the second one is PyTables-based. To distinguish between the two GTrackCore versions, from here on out the memmap version will be referred to as the 'old GTrackCore', 'old version' or 'memmap version', while the new implementation with PyTables will be referred to as the 'new GTrackCore', 'new version' or 'PyTables version'. If neither one is specified in a given setting, it means that both versions are equal implementation-wise.

The GTrackCore has, as briefly mentioned in section 2.2.6 on page 11, two main features: *track preprocessing* and *data retrieval*. Track preprocessing is the task of converting, or processing, textual genomic track data into binary data. Data retrieval is the task of retrieving the processed data. It is worth noticing that data retrieval is a more performance-critical feature, as it is performed several times, while the preprocessing is usually only executed once. As a consequence, the retrieval part of the GTrackCore has been subject to some sizable – and numerous minor – optimizations throughout the development stages of this work. Some of the larger ones will be addressed during this chapter, while others are left out.

5.1.1 Central concepts

In order to approach some of the implementation details in the following sections, a few of the central concepts and classes in GTrackCore will briefly be explained here. Other concepts will be introduced later, or further explained, as they become relevant.

Track A `Track` object represents a single genomic track, and is uniquely identified by its name and genome. The name of a track is a list of strings, which is its associated categories and a name. E.g. *DNA structure:Bendability* (colon-separated).

Track view The track view is the interface against the persisted binary track data. It is represented with the `TrackView` class. A more comprehensive description of this class will be given in section 5.6 on page 38.

Track attribute A track attribute is one of the columns of a track. Examples of track attributes are *starts*, *values*, or any other specified attribute. These are not restricted to the eight reserved core attributes of the GTrack format.

Genome region A genome region is used to identify a region of the genome, using a genomic interval. It contains the attributes *genome*, *chr* (sequence identifier), *start* and *end* base-pair, and *strand* (optional).

Genome element The `GenomeElement` class is a generic container, which is used to hold all the values associated with one genome element, e.g. one line with values from a textual track.

Track element Track elements are, at an abstract level, equal to genome elements in the sense that they both represent one element in the track. Semantically, the difference between a `GenomeElement` and a `TrackElement`[1] is that the former refers to an element which has not yet been preprocessed, while the latter is a member of the processed data.

When, e.g. iterating through the elements of a track view, objects of the type `TrackElement` are yielded.

Bounding regions A set of regions for where a track is defined to contain information. The bounding regions of a track are stored with the following information:

`chr` A sequence identifier for the bounding region, e.g. a chromosome.

`start` The start base-pair position of the bounding region.

`end` The end base-pair position of the bounding region, in end-exclusive notation.

[1] `TrackElement` is called `PytablesTrackElement` in the PyTables version.

`start_index` A positional index in the binary data of the first element that is encapsulated by the bounding region.

`end_index` A positional index in the binary data of the last element that is encapsulated by the bounding region, in end-exclusive notation.

`element_count` The number of elements within the bounding region.

5.2 A brief look at the preprocessor

Before one can retrieve binary track data from a genomic track using GTrackCore, the track has to be converted from textual to binary data. This task is done by the *preprocessor* of GTrackCore. The preprocessor can, at a high level, be divided into two phases: preprocessing *with overlaps* and preprocessing *without overlaps*. If a track being processed is *dense*, the first phase is skipped. If a track is *sparse*, both phases are performed.

Many of the operations of the HyperBrowser are performing more efficient when the elements of a track are not overlapping. For some operations it is also required mathematically, e.g. when calculating the base-pair coverage of a track. For sparse track types, two distinct datasets of binary data are created. One where possibly overlapping elements are preserved, and another where overlapping elements are merged.

With overlaps For sparse track types, the first phase is to process the track, preserving all the elements from the original track. Here, the textual file is read, line by line, continuously generating `GenomeElement` objects that are inserted into the dataset as binary data. If the track type of the data is any of the dense types, this phase is skipped.

No overlaps Both sparse and dense tracks are preprocessed without overlaps. Depending on whether the previous step, with overlaps, has been executed, this step either iterates the `TrackElement` objects from the previously generated binary data, or read lines directly from the textual track file. Again, the task is to generate `GenomeElement` objects that are inserted into the dataset as binary data. The dataset produced by this phase will never contain two overlapping track elements. For sparse tracks, a module of GTrackCore merge the overlapping elements into one continuous element.

Sorting and creating bounding regions For each of the previously mentioned phases, the datasets are sorted and accompanying bounding regions are created. The track elements for each of the datasets are sorted based on sequence id (e.g. chromosome), start base-pair and end base-pair, depending on whether the track has the attributes *start* and *end*. If neither start nor end is present, the track does not need to be sorted, as it already is so by the track type definition.

Finalization In the PyTables version, finalization is performed at the very end of the preprocessing. Here, the two datasets are merged into a

single PyTables database file, and track metadata is persisted within the database.

5.3 The preprocessed data

The preprocessed data is, as mentioned, divided into two completely distinct and independent datasets of binary data. One for track elements where overlaps may occur, and one without overlaps. Additionally, these datasets are sorted based on the seqid, start and end columns.

5.3.1 Multiple memmap files

In the old version of GTrackCore the binary data is stored as memmap files, where each file is one of the attributes of the track. These files are stored on disk in a specific directory structure that corresponds the bin-size[1], genome, overlap rule, and name of a track, following the format

```
/bin-size/overlap_rule/genome/track_name/memmap_files
```

where *memmap_files* are one or more files, *overlap_rule* is either `withOverlap` or `noOverlap`, and *track_name* is split into multiple directories according to what categories the track belongs to.

Bounding regions The bounding regions of a track are stored in a Python shelf [38], which is a dictionary-like object persisted as a file (with `.shelve` extension). The values of shelves can essentially be any Python objects. The shelf is persisted in the same directory as the memmap files.

5.3.2 A single PyTables database file

The new GTrackCore stores the binary data using a PyTables database, with one dataset, or table, for *with overlaps* and one for *no overlaps*. The tables have columns corresponding the attributes of a track. The database file is stored on disk in a specific directory structure, following the format:

```
/genome/track_name/database_file
```

The *track_name* is split into multiple directories according to which categories the track belongs to, similarly as before. Within the database file one finds almost the same structure, but with *groups* and not directories. The tables are located in the following path:

```
/genome/track_name/overlap_rule/table
```

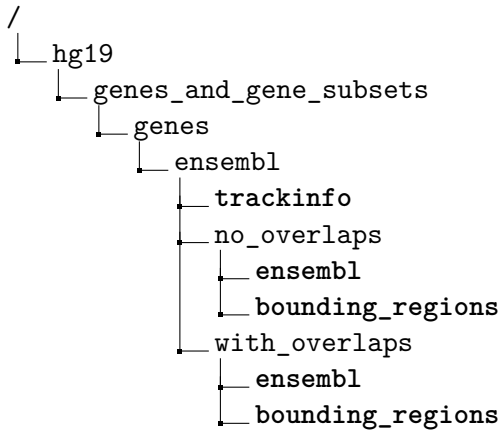
Here, all nodes but the last are groups, which are equal to directories, while the last is a PyTables Table node. Using the track *Genes and gene*

[1] The bin-size of a track defines how it is indexed (see section 5.7.1 on page 43). This only applies to the old version.

subsets:Genes:Ensembl as an example, we see that the database file is located in the directory path:

```
/hg19/Genes and gene subsets/Genes/Ensembl/ensembl.h5
```

The example following this paragraph shows the inner structure of the database file, and how the nodes are hierarchically stored, equal to a directory structure. The base node is a PyTables root node, while each following node is either a group or a dataset (marked with bold text). All the datasets are Table nodes, except for **trackinfo**, which is a File node. All node names have been normalized, restricting them to only consist of alphanumeric[1] characters, following the natrual naming convention of PyTables [31].



Bounding regions As seen in the previous database structure example, each of the overlap rules has a table for the track data, and another table for its accompanying bounding regions.

Chunkshape of the datasets

For all the datasets that are created by the preprocessor, the chunkshape of the dataset is automatically set by PyTables. The shape is set based on the expected number of rows that are put into each dataset, which is provided to PyTables upon the creation of the data tables.

5.4 Retrieving the binary data

The binary data is retrievable through an interface called a *track view*, which is represented by the `TrackView` class. To create such a view one first has to specify which genomic track one wants a view over. This is done by creating an object of the class `Track`. This class has an attribute called `trackSource`, which is the source from where one retrieves the track data. Furthermore, one calls the `Track.getTrackView` method to create a `TrackView` object.

Whenever one wants to retrieve any data from a track, the following information is required:

[1] a-z, A-Z, 0-9 and _

track_name The full name of the track, comprised of the categories the track belongs to and the name itself.

allow_overlaps A boolean value describing whether to retrieve data *with overlaps* or with *no overlaps*, i.e. the overlap rule.

genome_region The region one wants a view over. It is required that the region is located within one of the bounding regions of the track. This object must include the following attributes:

genome The genome this region is contained in.

chr The sequence identifier for the region (e.g. a chromosome name).

start The position of the start base-pair of the region (inclusive).

end The position of the end base-pair of the region (exclusive).

strand Which strand of the track to use (optional).

```
def get_track_view(track_name, allow_overlaps, genome_region):
    track = Track(track_name)
    track.addFormatReq(TrackFormatReq(allowOverlaps=allow_overlaps))
    return track.getTrackView(genome_region)
```

As seen in the example method `get_track_view` above, the classes `Track` and `TrackFormatReq` are used to get track views. `TrackFormatReq` is mainly used to specify whether to use the track data that includes overlaps or not. Figure 5.1 shows a simplified graphical representation of the call flow when creating track views.

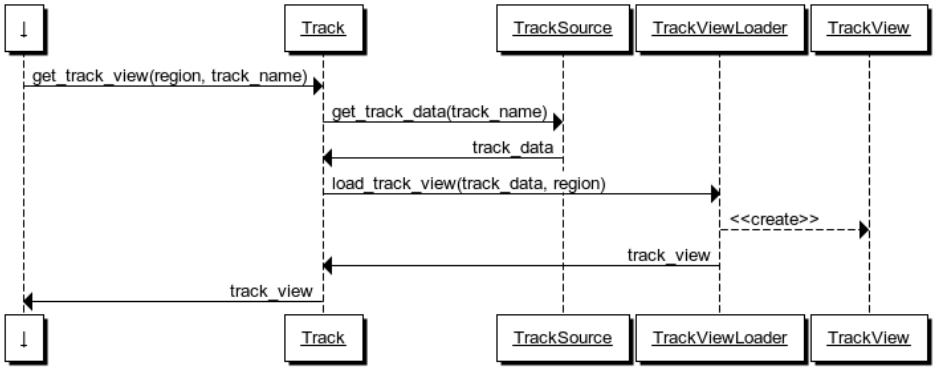


Figure 5.1: A simplified UML sequence diagram showing the flow of how track views are created.

5.5 Track data

Track data is a collection of data from a genomic track, and is stored in a Python dictionary with each of the columns in the track as values, and the accompanying column names as keys. The column values may be any type

of list containing the binary data. As genomic tracks often are very large, the lists are normally not read from disk when the track data dictionary is instantiated. Rather, a lazy approach has been implemented for both the old and new GTrackCore, with `SmartMemmap` and `VirtualTrackColumn` objects, respectively. The two next sections presents these two variants of list types that are used in the track data.

5.5.1 SmartMemmap

A `SmartMemmap` object represent one of the attributes of a track, and serves as a layer on top of the NumPy memmap. It utilizes the sequential reading of GTrackCore by caching portions of the memmaps, reducing overall memory consumption. Objects of this type contains the associated memmap filename and data type information of the track column.

The class implements two special Python double underscore methods for retrieving information from the binary data: `getitem(i)` and `getslice(i, j)`, which retrieves either a single item or a slice of data, respectively. The first is mostly used when iterating track elements in a track view, retrieving single data values, while the second is used when calling the 'as NumPy array' methods of a track view (see section 5.6.4 on page 41).

When either of the two methods are called, the `SmartMemmap` object caches a NumPy memmap object for the requested region, if it has not already been cached. `SmartMemmap` objects caches memmaps of approximately a million elements (2^{20}), where each memmap is called a *bin*. If the indices passed to `getslice` crosses the boundaries of a bin, a new memmap object is always instantiated with the offset to the requested region.

5.5.2 VirtualTrackColumn

A `VirtualTrackColumn` object, like `SmartMemmap`, represents one of the attributes of a genomic track. It contains a database connection to its database file, its associated table name and column name, and start and end indices referring to positions in the track table. At initialization, objects of this type do not hold any track data, as they load data lazily.

The class is designed to pose as a NumPy array, which means that any method that can be called for a NumPy array object, can also be called for a `VirtualTrackColumn` object. To achieve this, the class is a subclass of `VirtualNumpyArray`, which has two features: A cached NumPy array (`cachedNumpyArray`), and an implementation of all the special methods for NumPy arrays (double underscore methods). All of these methods are implemented using the following two steps:

1. If the `cachedNumpyArray` is *None* (not yet been read from disk), read a NumPy array by calling `as_numpy_array`, and then cache it.

2. Delegate the special NumPy method to the cached NumPy array.

This type of design is called a delegation pattern. The `as_numpy_array` method (see below) is defined in the subclass `VirtualTrackColumn`, and

does the actual reading from the database table. On line 5 in the method, the data is read from the table. When slicing `column`, a PyTables `Column` object, the special method `__getslice__(start, stop)` is called.

It is implemented to read the data in the table belonging to `Column` object, from *start* to *stop*, and returns it as a NumPy array object. Additionally, it is worth noting that even though the database reader object is opened on line 2, and closed on line 6, it does not mean that the database connection is actually opening and closing. This will be further explained in section 5.8 on page 50.

```
1 def as_numpy_array(self):
2     self._db_reader.open()
3     table = self._db_reader.get_table(self._table_node_names)
4     column = table.colinstances[self._column_name]
5     result = column[self._start_index:self._end_index:self._step]
6     self._db_reader.close()
7     return result
```

5.6 TrackView

The `TrackView` class serves as the interface between a user and the track data. There are two methods for retrieving data from a track view: retrieving one of the columns of a track as a NumPy array, or iterating through the track elements as `TrackElement` objects. A track view only encapsulates the track elements that were found by the *track view loader* (5.7 on page 43), i.e. the track elements that are covered by the requested genome region. When retrieving data from the track view, one can only get data from within this region. The `TrackView` is completely backwards compatible. From the perspective of a user the `TrackView` class is used in the exact same way in the new GTrackCore as in the old. The `TrackView` class has these noteworthy attributes:

Track attributes The attributes, or columns, of a track are the same lists as the values in the track data, with the exception that their start and end are restricted by the requested genome region. The attribute lists are passed to the track view from the *track view loader* upon initialization.

Genome anchor The genome anchor is used as an anchor for the *start* and *end* columns. This means that these columns are always relative to the anchor. The start of a track view thus always starts at the base-pair position 0 and not necessarily at the start of the requested region. The genome anchor attribute is the same as the requested genome region object.

5.6.1 Key features of a track view

Lazy loading

A track view read track data from disk lazily. It is only when the user requests the actual data of an attribute that the data is read, i.e. by calling the method for retrieving an attribute as a NumPy array. When using memmaps (e.g. `SmartMemmap`s), the read operation is even more lazy, meaning that the data is not read until it is used in some calculation or operation.

Handling points and partitions

Even though not all track types have both *starts* and *ends*, it is beneficial to let the tracks that only have one of them, imitate having both. This is because operations in the HyperBrowser are easier when all tracks can be viewed as Segments tracks. If a track has type *Points*, it only contains *starts*. The accompanying *ends* to the *starts* of a track can easily be given by adding 1 to all the *starts*, since Points, by the track type definition, have length one. In the opposite case, where a track only contain *ends*, the *start* of a track element is the *end* of the preceding element. This is because tracks with ends but not starts are defined to be dense. During the preprocessing of such tracks, an extra track elements is added at the beginning of every bounding region in the binary data, only to be used as start for the actual first element.

Counting elements

Upon initialization of a track view, a count of the number of elements the view encapsulate, is performed. This is done differently based on the track type. If the track representation is dense, i.e. the track has a track element for every base-pair position, the number of elements is equal to the length of the region the view covers. Otherwise, for the memmap version of GTrackCore, the count is performed by reading one of the lists of the view from disk as a NumPy array, and then getting the number of elements from the NumPy object. A method in the `TrackView` class is used to exclude blind passengers (section 5.6.3 on the following page) if the track type is sparse. For the new version, the count is done by finding the length of one of the `VirtualTrackColumn` objects, and than iteratively subtracting the number of blind passengers for sparse tracks.

5.6.2 Differences between the old and new version

Upon initialization of a track view, a variable denoting whether to use PyTables functionality or not is set (`should_use_pytables`). This variable is used internally to determine how to perform certain tasks. It is set by checking whether or not the list objects passed to the track view are of the new `VirtualTrackColumn` type. If they are not, i.e. any other list structure (e.g. `SmartMemmap`), the track view is still fully functional. The following

tasks has to be handled differently when the track view is using `VirtualTrackColumn` objects as the attributes of the track.

Iterating through track elements

In the old version, the `TrackElement` object has an index variable, which is being incremented for each iteration in the track view. When retrieving the data from the track element, the index is used to determine where in a track attribute, i.e. the `SmartMemmap` object, to retrieve the data from.

In the new version, we utilize the highly optimized iteration tools provided by PyTables. Given the known start and end indices of the track view (found by the *track view loader*), we simply create an iterator for `Table` class of PyTables, with the start and end indices. The `PytablesTrackElement` class has a pointer to the current *row* in the track table. For each of the columns in the track, one can retrieve the associated data from the PyTables Row object.

Handling points and partition

In order to pose having *starts* and *ends* for tracks without starts and ends, respectively, the method `handlePointsAndPartitions`[1] is called. If the track type is either *Points* or *Partition*, *ends* or *starts*, respectively, is "imitated" to be present.

The method for how this is handled differs slightly for the old and new GTrackCore. One of the steps of this procedure involves changing the start or end index of the list-structures in the track data. For `SmartMemmap` objects this is done with Python-slicing, which only moves pointers in the underlying memmaps. When using `VirtualTrackColumn` objects, the data is read from disk when slicing (previously explained in section 5.5.2 on page 37). To ensure read laziness, `VirtualTrackColumn` has a method to avoid reading data when doing "slicing", called `update_offset`, which updates the start and end indices of the object.

5.6.3 Blind passengers[2]

For sparse track types, it is possible that the track view loader creates track views that include track elements that should not be included. The track view loader finds the first track element that is "touching" the requested genome region (explained in 5.7.2 on page 44). This does not guarantee that there are no elements that has a *start* that is after the first track element and an *end* that is before the start of the requested genome region. Such track elements are called *blind passengers*, and are ignored when retrieving

[1] `handle_points_and_partitions_for_pytables` when using `VirtualTrackColumn` objects

[2] The term *blind passenger* refers to a passenger, or elements in the setting of genomic tracks, that should be excluded in a given setting, while it is not. The term is a direct translation of the Norwegian term for *stowaway*, which is a person who hides aboard a ship or other conveyance in order to obtain free passage.

the track data. Figure 5.2 shows an illustration of blind passengers for a sparse Segments track.

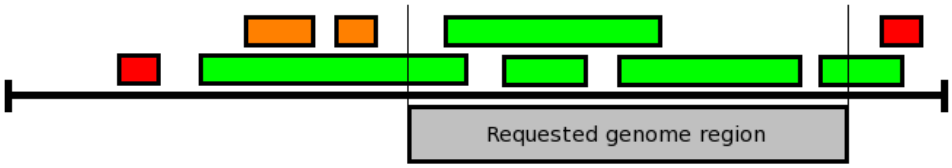


Figure 5.2: **Illustration of blind passengers for a sparse Segments track.** The black line is a genomic sequence. The boxes above the line are track elements, and the box underneath is the requested genome region. The green boxes should be included by the track view, the red should not be included, while the orange ones are blind passengers that are falsely included.

5.6.4 Examples of retrieval

In the two following sections, an example of the two retrieval methods are presented. The examples show how one can use GTrackCore to perform a simple track operation: Find the sum of all the values in a track. The first example solves the task using an 'as NumPy array' method, while the second iterates through all the track elements. Both methods calculate the sum for a specified track, overlap rule and set of genome regions.

The 'as NumPy array' methods

For all the reserved columns in the GTrack format, there is a method in the `TrackView` class in the form `<core attribute>AsNumpyArray()` (e.g. `startsAsNumpyArray()` or `edgesAsNumpyArray()`). In addition, there is one method called `extrasAsNumpyArray(key)`, which is used to retrieve NumPy arrays from an extra attribute of a track. The *key* specifies which extra attribute. These methods are used to retrieve data from one attribute as a NumPy array.

The following method is an example of how one uses the 'as NumPy array' methods in a track view. On line 6, a NumPy array, containing the values, is retrieved from the track view by calling the `valsAsNumpyArray` method. It is only the values that are encapsulated by the track view that are retrieved. Blind passengers are removed from the NumPy array before it is returned from the track view.

```
1 def sum_of_values(track_name, allow_overlaps, genome_regions):
2     value_sum = numpy.float128(0)
3     for region in genome_regions:
4         track_view = get_track_view(track_name, allow_overlaps,
              region)
5         vals = track_view.valsAsNumpyArray()
6         value_sum += vals.sum()
7     return value_sum
```

Iteration

While retrieving data from a track view using the previous method almost always is the quickest by far, some times the only option is to iterate through all the track elements, e.g. when iterating through the edges of a track. When prototyping new functionality, it may also be easier to work with an object that includes all the attributes of a track element, rather than calling the 'as numpy array' method for each attribute. For the sake of comparison, the iteration example is still to find the sum of all the values in a track. On line 6, in the example below, is an iteration through all the elements of a track view. For each of the track elements, the *value* field is retrieved. When using the track view iterator, blind passengers are skipped continuously by the iterator.

```
1 def sum_of_values_iter(track_name, allow_overlaps,
      genome_regions):
2     value_sum = numpy.float128(0)
3     for region in genome_regions:
4         track_view = get_track_view(track_name, allow_overlaps,
              region)
5         for track_element in track_view:
6             value_sum += track_element.val()
7     return value_sum
```

As a small-scale benchmark test, to illustrate the running time difference, the two functions have been run with a Valued Segments track with one million track elements, for one genome region. This test was performed using the PyTables version of GTrackCore. When using the iterator, the time it took to sum all the values in the track was 2.66 seconds, while it only took 0.33 seconds when retrieving the values as a NumPy array, and summing these using the NumPy method *sum*.

5.6.5 GraphView

While the `TrackView` is the main interface to the persisted data, it can be beneficial in settings when working with graph data, i.e. tracks with interconnection, to use an interface that is tailored for that specific need. The `GraphView` interface allows one to walk the edges and nodes, i.e. the track elements, of a graph, in the order they appear in for each node, and not the order of the track elements. This type of iteration basically means reading track elements that are randomly placed in the dataset, i.e. not necessarily consecutively.

The implementation of the graph view in the PyTables version of GTrackCore is very similar to the old one, with the exception that we look up a PyTables Row object in the table based on the provided row index, rather than directly retrieving the requested values from the NumPy arrays based on the same index. By doing so, we never cache potentially large NumPy arrays when one only need a single value from the dataset.

5.7 Creating TrackView objects

We have intentionally followed the same pattern for creating `TrackView` objects in the new GTrackCore as in the old one. This pattern is illustrated in Figure 5.1 on page 36. This decision was made so that the GTrackCore can be used in the exactly the same way from the outside as before. Since the new version is based on a database, rather than using memmap files, the methods for creating the track views differ a lot. The next section takes a brief look at how the creation of track views is done internally in the old version (5.7.1). Afterwards, a detailed description for the new version is given (5.7.2). Details such as specifying genome and overlap rules will be left out in the explanations, but are a necessity to create `TrackView` objects.

Note that both versions are designed to load data as lazily as possible, meaning that no data should be read before it is used. The reason for designing the track views to read data lazily is solely for performance. The data is persisted on disk, and disk read/write operations are, as commonly known, very costly. If a Function track, for instance, has five extra attributes, but the only interesting attribute for a certain operation is the value attribute, a considerably amount of time would have been spent reading the five extra attributes from disk if data was read in advance, rather than lazily. When taking very large tracks (millions or even billions of elements) into account, this would be potentially highly wasteful use of time and resources. It would probably also imply certain memory issues, as the tracks can contain several gigabytes of data.

5.7.1 Memmap version

From the `Track.getTrackView` method, the first step is to obtain a collection of the attributes from the track, i.e. the *track data*. The method `getTrackData` in `TrackSource` is called to retrieve this data. Here, one loops through all the files in the directory of preprocessed data of a track. For each file that is a memmap file, a `SmartMemmap` object is created, with a link to that specific memmap file, and adds it to the track data collection. This object now acts a view over *one* of the attributes of the track, e.g. all the *starts* or all the *values*. The track data collection is a dictionary where the keys are the attribute names with the corresponding `SmartMemmap` objects as the values.

The next step is to call `loadTrackView` in `TrackViewLoader`, passing along the track data from before, and the genome region one wants a view over. At this point, all the `SmartMemmap` objects are views for the for the *whole* track for each attribute of the track. The goal of `loadTrackView` is to restrict these views to only include what that is covered by the given *genome region*. At a technical level, this means changing the start and end pointers of the memmaps that are connected to from each `SmartMemmap` object, which is done by slicing the memmap. To slice the memmap, one need to know where to slice it, i.e. one need to find the start and end position in the memmaps that corresponds to the genome region. In order to find the memmap indexes, the old version of the GTrackCore uses an ad-hoc

indexing method.

leftIndex and rightIndex

In addition to all the memmaps that represent the attributes of a track, an additional two memmaps are created when the track is being preprocessed. These two memmaps, called *leftIndex* and *rightIndex*, are indices for the base-pair positions of a track, and are used to ensure quick loading of track views.

The *leftIndex* is an array, where the 0th position represent all the base-pair positions from 0 to 100000, the 1st position represent all the base-pair positions from 100000 to 200000, and so forth. For each of these positions, the array contains a value, which is a positional index for all the track attribute memmaps. This is the index in the attribute memmaps of the first occurrence of the first element with start base-pair between the mentioned base-pair positions in the memmaps. The *rightIndex* works in a similar fashion, but is an index for the end base-pairs rather than the starts. Note that this indexing method may result in track views that include more track elements than there actually are within the requested genome region. These extra elements are later removed sequentially from both ends when the track view is created by calling the `sliceElementsAccordingToGenomeAnchor` method.

When the `SmartMemmap` objects have been sliced according to the indices that were found from the *leftIndex* and *rightIndex*, a `TrackView` object is created with the `SmartMemmap` objects passed along to it.

5.7.2 PyTables version

Since the high-level structure in the PyTables version is the same as in the old version, all the steps that are described in the following sections are the same as in 5.7.1. The method for how we have achieved the same goal is, however, very different.

Get track data

The first step for creating a `TrackView` is, as before, to call the method `get_track_data` in the `TrackSource` class. Here, a database connection to the HDF5 file is created and opened. All the column names, i.e. the attribute names of the track, are collected from the track table, along with the number of elements in the table. For all the columns in the table a `VirtualTrackColumn` object is created, each representing an individual column in the table. Upon creation they are passed their column name, a database reader object, and start and end indices in the table. The indices are at this point set to cover all the elements in the table, as the requested genome region has not yet been specified. The `VirtualTrackColumn` objects are put in the track data dictionary with the corresponding attribute names as keys, similar to the old version and the `SmartMemmap` objects. The dictionary is then returned to `Track.getTrackView` method, and subsequently passed to

`load_track_view` in `TrackViewLoader`, along with the *genome region* for which one want to view the track.

During the lifetime of a `Track` object, each of the columns in the track data dictionary (`VirtualTrackColumn` objects) is only instantiated once, as in the old GTrackCore. I.e., if multiple track views are loaded from the same track, the dictionary is only created once.

Track view loader

As mentioned, the `VirtualTrackColumn` objects in the track data dictionary are upon creation set to cover all the elements in the track table. The main responsibility of the `load_track_view` method is to restrict the columns to only cover the elements that are within the requested genome region. A track element is considered to be within a genome region if the element is "touching" the region. That an element is touching a region means that it either is completely enclosed by the region or that at least some of its base-pair positions are overlapping with the region. As a consequence, the first and last elements of a track view may partly be outside the requested genome region. An example of this can be viewed in figure 5.3.

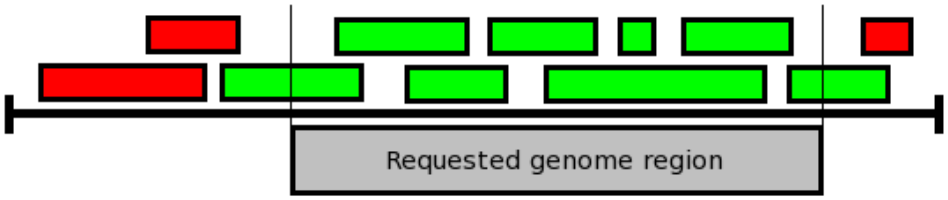


Figure 5.3: **Illustration of touching track elements for a sparse Segments track.** The black line is a genomic sequence. The boxes above the line are track elements, and the box underneath is the requested genome region. The green boxes are defined to be *touching* the requested genome region, while the red ones are not. The touching track elements are included in the loaded track view. The first and last of the green track elements are partly overlapping with the requested genome region, and are thus also included in the track view.

To find the correct indices in the track table, we call the `start_and_end_indices` function in the `IndexRetrieval` module, which will be explained in the next section. When the correct indices are found, each of the offsets (start and end indices in the track table) of the `VirtualTrackColumn` objects are updated so that they now only cover the elements for the given region. Next, a `TrackView` object is instantiated with the `VirtualTrackColumn` objects.

Index retrieval

The `start_and_end_indices` method retrieves the exact indices for where a given genome region starts and ends in the table data[1]. Because no track

[1]Blind passengers may still occur (see figure 5.2 on page 41).

view can cross the boundaries of a bounding region, the first natural step of retrieving the start and end indices, is to query the bounding region table to find an enclosing bounding region. A bounding region is enclosing a genome region if the following requirements are met:

- The *sequence id* for the bounding region and the genome region are equal.
- The *base-pair start* of the bounding region is equals to, or less than, the *start* of the region.
- The *base-pair end* of the bounding region is equals to, or greater than, the *end* of the region.

If no such enclosing bounding region exist, the requested genome region is not valid, and an empty track view is returned. In the opposite case, the start and end indices for the bounding region in the track table are retrieved. At this point, we have reduced the problem area from covering the whole track table, to only encapsulate the range of one bounding region.

Underneath is a simplified version of how the bounding region table is queried. The *where* statement in a PyTables Table node returns an iterator of the resulting rows. When looking up an enclosing bounding region, the resulting set is always of size zero or one, as the bounding regions are not allowed to overlap.

```
query = '(chr == region_chr) & (start <= region_start) &
         (end >= region_end)'
condvars={'region_chr': genome_region.chr,
          'region_start': genome_region.start,
          'region_end': genome_region.end}

br_table.where(query, condvars=condvars)
```

When, and if, an enclosing bounding region is found, the next step is to find the correct indices within the region. Depending on track type, one of the two techniques are used for finding the indices:

1. If a track contains neither starts nor ends, and has defined track elements for every base-pair position of all the bounding regions, the procedure of finding the indices is a trivial mathematical calculation.
2. If the track contains either starts or ends (or both), a search for the correct indices in the track table is required.

1. Neither starts nor ends When a track is of a type without starts and ends, there is a track element for every base-pair position within the bounding regions of the track. Because of this, one can use the start and end base-pair positions that are given with the genome region and the found bounding region to calculate which indices in the track table that cover the requested genome region.

In the following calculation, start index is the offset of the genome region in base-pair positions from the the start index of the enclosing bounding region, and end index is the length of the genome region (`end_bp` − `start_bp`) as offset from the start index. Following is a simplified version of the calculation of first the start index, then the end index:

```
start_index = br.start_index + (gr.start_bp - br.start_bp)
end_index = start_index + (gr.end_bp - gr.start_bp)
```

In the calculations above, *br* is the already found bounding region and *gr* is the requested genome region. The variables with *index* suffix refers to indices in the track table. The *bp* suffix means base-pair position.

2. Either starts or ends When a track has either starts or ends, the track may have track elements that cover multiple base-pair positions, and there may be gaps between them. The calculation in the previous paragraph is therefore not applicable. To find the correct indices in the track table, a search for the correct region is necessary.

The trivial way to perform this is a sequential search, iterating through all the track elements within the bounding region. When the first track element that is touching the requested genome region is found, the start index is found. Thereupon, the iteration will continue until the first track element that is outside the genome region, denoting the end index. For large tracks, this kind of search could be very time consuming.

The next two sections will address how the index search is implemented in the PyTables version. The first includes our first attempt, which was to query the track table to find the correct start and end indices, while the second try uses a binary search algorithm.

Using queries to retrieve the indices

PyTables includes strong functionality for indexing columns and querying them efficiently, as they put it themselves:

> *One characteristic that sets PyTables apart from similar tools is its capability to perform extremely fast queries on your tables in order to facilitate as much as possible your main goal: get important information *out* of your datasets.*
>
> *PyTables achieves so via a very flexible and efficient query iterator, named Table.where(). This, in combination with OPSI, the powerful indexing engine that comes with PyTables, and the efficiency of underlying tools like NumPy, HDF5, Numexpr and Blosc, makes PyTables one of the fastest and more powerful query engines available.*
>
> — *[1]*

To utilize these strengths, we implemented methods that queried the track table to retrieve the correct start and end positional indices in the track table. To improve the query execution time, we indexed the start and end columns in the track table, as it is these attributes that are used to find

the indices. This is done by simply calling the `create_index` method of the PyTables Column object. The PyTables query engine finds out in runtime whether or not it can use a column index for a given query. The queries we created retrieved all the positions of the track elements in the track table that touched the genome region. From the query result – a list of index positions in the track table – we extracted the first and last elements, which was the correct start and end indices.

The code snippet in Listing 5.1 shows how we queried the table, using a Segments track as an example. The object `genome_region` is the request genome region, `table` is the PyTables Table node object, and `br_start` and `br_stop` are the bounding region indices, which are used to restrict the search area. On line 1 is the query for finding all the track elements that are touching the genome region for a Segments track. Next are the `condvars` (conditional variables) that are injected into the query by PyTables. On line 5, we see how the table is queried, using the *where* statement. As mentioned, we are only interested in the indices retrieved from the query, and not the data within the row objects. Consequently, we used the PyTables `get_where_list` method, which retrieves the result indices, rather than the traditional `where` method, which retrieves an iterator for the resulting PyTables Row objects. From the resulting list, we returned the first and last index elements, i.e. the correct start and end indices.

```
1  segment_query = '(end > region_start) & (start < region_end)'
2  condvars={'region_start': genome_region.start,
3            'region_end': genome_region.end}
4
5  region_indices = table.get_where_list(segment_query,
6                   start=br_start, stop=br_stop, condvars=condvars)
7
8  # start_index, end_index
9  return (region_indices[0], region_indices[-1] + 1)
10        if len(region_indices) > 0 else (0, 0)
```

Listing 5.1: A simplified version of how to retrieve the start and end indices when using PyTables queries.

This seemed to be an efficient method, but – as it later turned out – only when working with either small track, or a small number of track views. The reason for why using queries was slow has to do with a "caching bug" in PyTables. The details around this will be addressed in section 6.2 on page 57. Nonetheless, we needed an improved solution. As all the track table data is already sorted, we completely removed column indexing, and implemented our own method for searching through the vast amount of data.

Binary search

Generally, database tables are not meant to be queried for where certain rows are located. Instead, the queries result in sets of data that fulfill some criteria. Following the design of the GTrackCore, we needed to find the start and end indices for where this criteria (the requested genome region) was

fulfilled. Given how the data in GTrackCore is stored, the criteria always produce results that are stored consecutively in the table, and can therefore only result in one region – the one we want the start and end indices of. To take advantage of the sorted data, we implemented an ad-hoc binary search algorithm. When performing the search, one can only check if a row is within or outside of a given region, but not whether the row is the first or the last that fulfill the criteria. We thus used the binary search to reduce the problem area from the bounds of the bounding region, which may be tens of millions elements, to a small range, about 1000 elements, in just a few comparisons. The binary search algorithm has a complexity of $\mathrm{O}\Big(log_2(n)\Big)$, resulting in relatively quick searches. With the reduced problem area, we finish the search using the trivial sequential search method, as described before. It is worth noting that this search has about $log_2(n)$ disk accesses, which can be cause of poor performance due to the slow nature of disk I/O.

The implementation The binary search algorithm, `improve_min_and_max_index`, is shown in Listing 5.2. As aforementioned, the task of the method is to refine the search area to a level where the sequential search method efficiently can be used. This occurs when the number of track elements is below the critical limit `ITERATION_THRESHOLD`, which is set based on how many rows that are cached by PyTables when iterating through them. [2] Depending on track type and whether the task is to find start or end index, the method is passed either the *start* or *end* as `column`, and the *start* or *end* of the genome region as `bp_position`. When, for instance, searching for the start index for a Segments track, `column` is set to *start* and `bp_position` is the *start* of the genome region. On line 4 and 5, the mid index is calculated and the accompanying row (track element) is read from disk. If the *start* value of the `row` is less than the *start* of the genome region, we know that the correct start index is located somewhere after the row. Consequently, we set the min index to be equal to the current mid index. In the opposite case, we know that the correct start index is before the row, and thus set the max index to be the current mid index. For each step, the search area is reduced by 50 %.

```
1  def improve_min_and_max_index(min_index, max_index, table,
2                                column, bp_position):
3      while max_index - min_index > ITERATION_THRESHOLD:
4          mid_index = min_index + ((max_index - min_index) / 2)
5          row = table[mid_index]
6
7          if row[column] < bp_position:
8              min_index = mid_index + 1
9          else:
10             max_index = mid_index
11
12     return min_index, max_index
```

Listing 5.2: The `improve_min_and_max_index` method.

The code snippet in Listing 5.3 shows how the start and end indices are found for a Segments track. The method parameter `gr` is the requested genome region, `table` is the track data table, and `br_start` and `br_stop` are the previously found bounds of the enclosing bounding region. On line 2, the bounds of the start index are restricted using the binary search. The initial bounds used were equal to the start and end indices of the bounding region. Next, the sequential search is executed, restricted by the found min and max indices. We know that the first occurrence of a track element that is touching the genome region is the current start index. This requirement is met when the criteria on line 5 and 6 is fulfilled. To extract the index in the table from a row object, the `nrow` attribute is used.

To find the end index, an equal procedure is performed. Since the end index naturally is positioned after the start index, we can use the newly found start index, rather than the start index of the bounding region, to restrict the initial bounds when performing the binary search. Additionally, the *end* of the genome region is passed to the search algorithm. As before, the next step is the sequential search. The criteria on line 13 is the first track element that is *not* touching the requested genome region, which is the correct end index (exclusive).

```
1  def start_and_end_indices_segments(gr, table, br_start, br_stop):
2      min_index, max_index = improve_min_and_max_index(br_start,
3                                br_stop, table, 'start', gr.start)
4      for row in table.iterrows(start=min_index, stop=max_index):
5          if (row['start'] < gr.start < row['end']) or
6                (gr.start <= row['start'] < gr.end):
7              start_index = row.nrow
8              break
9
10     min_index, max_index = improve_min_and_max_index(start_index,
11                                   br_stop, table, 'start', gr.end)
12     for row in table.iterrows(start=min_index, stop=max_index):
13         if row['start'] >= gr.end:
14             end_index = row.nrow
15             break
16
17     return start_index, end_index
```

Listing 5.3: A simplified version of how to retrieve the start and end indices when using binary search and sequential search.

Note that this example is somewhat simplified and that not all special cases are accounted for.

5.8 The database interface

To keep the GTrackCore code base tidy, we implemented a database interface to use against PyTables. This interface includes common methods that are used from various parts of the GTrackCore. It contains an abstract base class called `Database`, and two subclasses: `DatabaseWriter` and

`DatabaseReader`. The writer class is mainly used by the preprocessor, and contains methods for manipulating the PyTables file, e.g. creating groups and tables. The `DatabaseReader` is used when one only wants to read from the database, e.g. when retrieving data. Each of the subclasses has their own *open* method that calls the open method in the superclass `Database` with the correct mode, i.e. *read* for the database reader and *append* for the writer. Append mode is equal to write mode, with the exception that the content of a file is kept, rather than overwritten, when opening an already existing file. The open method in the reader class also checks if the database file is already open before calling the open method in the base class. In addition, the open and close methods in the `Database` class handle locks for the PyTables files. Figure 5.4 show a somewhat simplified class diagram of the involved classes.

The stored data is organized, as mentioned in section 5.3 on page 34, in groups and datasets, according to the genome, name and overlap rule of a track. At a generic level, groups and datasets are referred to as nodes. These nodes are retrievable by their paths, i.e. a list of the nodes to a specific node. A set of functions for finding such node paths, based on their genome, name and overlap rule, are located in the module `NameFunctions` in the `util/pytables` package in the code base. Once a node path is known, it is retrievable via the `Database` method `get_node(node_name)`, or `get_table` for tables. These methods call the `get_node` method in the PyTables package, and caches the nodes for later usage.

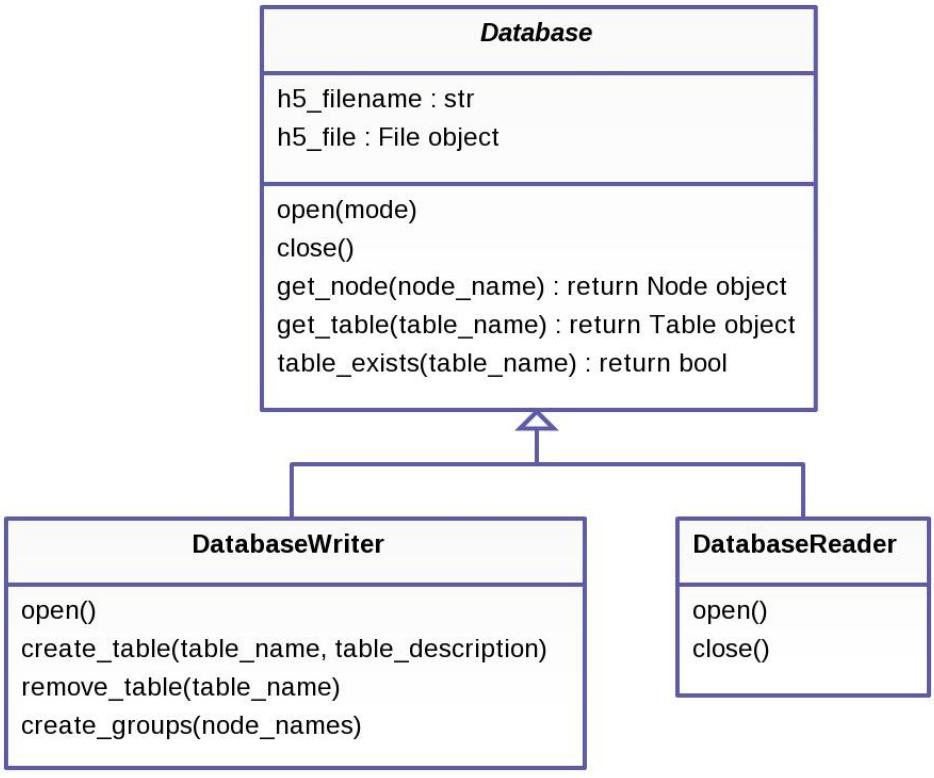


Figure 5.4: A simplified UML class diagram of the database interface.

5.8.1 Open database connection

Opening and closing files are, to a certain degree, costly operations, and proved to provide large overheads for operations with many bins, i.e. track views. Because of this, we implemented a database reader connection that is kept open until the Python process is terminated. A Python module called `atexit` lets us call a method right before the process ends, called `close_pytables_files`, which closes all the remaining open PyTables files. The `atexit` module also lets us persist metadata at the end of a run. The result of this optimization is presented in section 6.2 on page 57.

When GTrackCore is used for preprocessing, `DatabaseReader` objects are also used for some of the tasks. Here, however, it is crucial to close these connections as one switches between using a reader and a writer at several occasions, for the various tasks that are performed when preprocessing tracks. The reason for why we have to close these connections is because one can not have a file (the HDF5 file) opened in both read and write mode. We thus made sure to always call the close method of the reader whenever needed from the perspective of the preprocessor, even though it might not be necessary when only retrieving data. The close method either actually closes the database connection – or does nothing – based on a preprocessing state variable (`gtrackcore.preprocess.is_preprocessing`), which tells us whether or not a track is being preprocessed. Hence, the close method of the `DatabaseReader` is as follows:

```
1 def close(self):
2     if gtrackcore.preprocess.is_preprocessing:
3         super(DatabaseReader, self).close()
```

5.9 From tables to arrays

At the end of the development process we decided to test the use of PyTables arrays as opposed to tables. Consequently, we had to implement functionality for creating datasets of array objects, where each array is equal to each of the columns of a table. PyTables has excellent support for this exact purpose. A NumPy array can be read from the `Column` object of a table, and array datasets can be created in the PyTables file, instantiated with a NumPy array, using any of the `File.create_array` methods. We decided to use the `File.create_carray` method because `CArray` objects are stored using a chunked layout, and are thus compressible. We did not need enlargeable arrays (`EArray`) as the table data already is preprocessed and thus constant. The code snippet below shows a simplified version of the creation of the arrays from the table data:

```
1 def create_c_arrays_from_table(table, array_group):
2     for column_name, column in table.colinstances.iteritems():
3         h5_file.create_carray(array_group, column_name,
4                               obj=column[:])
```

On line 2, we iterate through all the `Column` objects of the table, while on line 3, we create a `CArray` object within the PyTables group `array_group`. The name of the array node is set to `column_name` and instantiated with the NumPy array retrieved from the `Column` object by slicing it (`column[:]`). The arrays are stored in the PyTables file under the group `column_carrays`, which is located at the same place as the track data table in the PyTables file structure.

5.9.1 Only the 'as NumPy array' methods

As using arrays instead of tables only is for testing purposes, it is only the 'as NumPy array' methods that has the possibility to read from arrays as opposed to table columns. This is done by providing the arrays as a data source in the `VirtualTrackColumn` objects, rather than table columns. All other parts of the code base that reads any data still uses the table as data source. This includes both the track view and the graph view iterator, and the index retrieval part of the track view loader.

5.10 Track operations

In order to compare the old and new version of GTrackCore, we created a set of typical track operations. All of the operations are separate methods that each take a track, overlap rule, and a set of genome regions as arguments. The genome regions used in the tests are typically the chromosomes for the genome of the track one performs the operations on, but can be any other set of regions. The track operations are located in the `tools/TrackOperations` module. The following list is an overview of the operations that were created to test the performance of the GTrackCore versions.

- *Base-pair overlap* for two Segments tracks
  - *NumPy arrays:* Retrieves data as NumPy arrays, and calculates the overlap using an efficient algorithm
  - *Iterator:* Retrieves data using an iterator, and calculates the overlap in a naive, straight-forward method
- *Sum of weights* for Linked track types with weights
  - *NumPy arrays:* Retrieves data as NumPy arrays and sums the values using the NumPy *sum* method
  - *Iterator:* Retrieves data using an iterator (the `GraphView` interface in GTrackCore), continuously summing the weights
- *Sum of values* for Function tracks. Retrieves data as NumPy arrays and sums the values using the NumPy *sum* method
- *Count of elements* for all types of tracks

Chapter 6

Code improvements throughout the project

In this chapter we present the largest code improvements made to the new GTrackCore that lead to the final result, i.e. the PyTables-based version of GTrackCore. Every improvement choice is explained along the way to shed light on the agile and evolving nature of the devolopment process, as described in section 3.1 on page 21. There are two types of improvements made to the code: *refactoring* and *optimizations*. The former is changes that increased the quality of the source code itself, while the latter has increased performance.

There were many other small changes, refactorings and optimizations to the new GTrackCore during development, than what is being mentioned in this chapter. These changes range from simply re-naming a variable or method, to moving entire modules to a more logical place in the project directories. Most of these changes can be found in the change log in the project repository (Appendix A on page 99).

6.1 Refactoring

As mentioned in section 3.1 on page 21, a considerable amount of the development process was spent refactoring the code to improve its overall quality. This section provides insight to the largest code improvement we did through the process of refactoring, i.e. the database interface. Moreover, we see a smaller improvement concerning the `VirtualTrackColumn` class.

6.1.1 The database interface

While the current interface against the PyTables file database is fairly simple with three classes and 150 lines of code (described in section 5.8 on page 50), it was at one point a large and complex structure with 13 classes and over 400 lines of code. Our initial design revolved around the *tables* in the PyTables file, i.e. for each connection there was *one* associated table. This design is seen in Figure 6.1 on the following page as a UML class diagram. All the classes above the green line were part of the initial design. Here, we

also included classes that we thought would be needed for future scenarios, e.g. the ones ending with `ReadWriter`. These were meant for cases where one needed append mode on a PyTables file, e.g. if one were to add more data to a dataset or edit the metadata of a track.

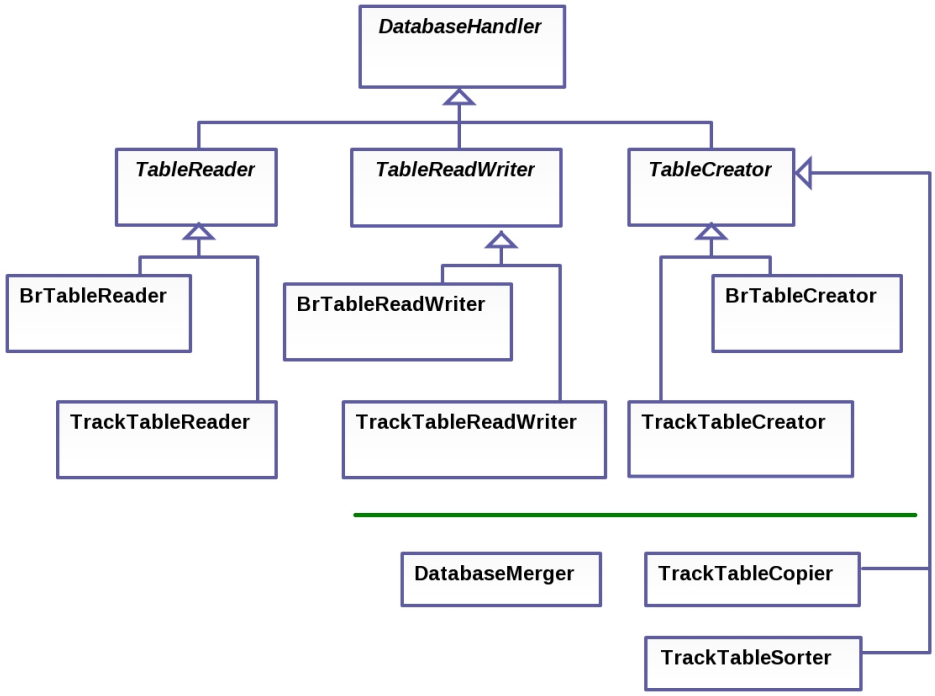


Figure 6.1: A simplified UML class diagram of the initial database interface. The classes below the green line were added later as new functionality was needed.

The classes below the green line in the figure were added because of new functionality that the old design was not facilitated for. Two of these classes was new functionality that was a result of two separate issues we had with PyTables. The first was the need to reshape a column, i.e. a multidimensional dataset, which is not possible for table columns in PyTables. The second was a slice-assignment issue with the first sorting algorithm we used. Both of these are described in detail by Rongved in [34].

Both of the issues were solved by creating a new table based on the old table. In other words, they both needed a connection to two tables; one in write mode and one in read mode. As mentioned earlier, it is not possible to have multiple connections opened to a single file if any of them are in write mode. Furthermore, the database interface was only facilitated to have *one* table per connection. Hence, we created two new classes within the interface that were equipped to work on two tables simultaneously: `TrackTableCopier` solved the reshape issue by creating a new table with columns of larges shape and then copying the data into the new table. `TrackTableSorter` solved the slice-assignment issue by copying the processed data to a new table, but this time in sorted order.

The resulting code proved to be complex, disorganized, and unreadable,

and further maintaining of the code would imply similar types of complex structures. This fact lead to the redesign of the database interface. Now, a database connection is simply an open PyTables file, in either read or write mode, with methods for manipulating said file. It is up to the modules that uses a database connection to retrieve the datasets, or nodes, it needs. The new functionality that inspired the re-design was all part of the preprocessing part of GTrackCore and was consequently moved to a module called `preprocess/pytables/CommonTableFunctions`. The database interface has, by being much simpler and more organized and general, improved the readability and maintainability of the code, while still being easy and convenient to use, and thus improved the overall code quality.

6.1.2 VirtualTrackColumn

One of the other larger refactorings we did was to improve the design of the the list-like objects used in the track data, i.e. the `VirtualTrackColumn` objects. These are designed to imitate being NumPy arrays by using a delegation design pattern, as described in 5.5.2 on page 37. The main reason for following this pattern was so that the `VirtualTrackColumn` objects could be used in the same way as `SmartMemmap` objects, i.e. exactly the same way one uses NumPy arrays. Another reason was to easier enable caching of track data. The previous version was called a `TrackColumnWrapper`, and was exactly that, i.e. a wrapper of the PyTables `Column` class. Whenever one retrieved data, a NumPy array was read from the track data table, and no caching of data was performed. Because one rarely retrieves the exact same data twice from a track view, at least in the scope of our tests, the caching did not have any significant impact on the performance other than using a small amount of more memory.

Making the `VirtualTrackColumn` class imitate being a NumPy array proved to reduce the amount of special case handling for some of the 'as NumPy array' methods in the `TrackView` class, as objects of this type could be handled in the exact same way as the `SmartMemmap` objects and NumPy arrays. Thus, less changes had to be made to the `TrackView`, keeping maintainability at a constant level.

6.2 Optimizations

There were two large optimizations throughout the project, related to the retrieval of data in GTrackCore, that improved the performance of the new version significantly: *index retrieval* and *open database connection*, both of which described in Chapter 5. This section points out these improvements, and illustrates them by using detailed profiles from running test operations with different configurations.

6.2.1 Index retrieval

As described at the end of section 5.7.2 on page 44, the search for the correct start and end indices when loading track views have been subject to some

optimization during the development. Our initial approach was to utilize the table queries of PyTables, with column indexing to reduce the query time. When using this approach, the PyTables version turned out to be very slow in benchmark tests, compared the old version of GTrackCore.

To investigate why the PyTables version was slower we profiled an execution of the *count of elements* operation with 33 167 track view loads for the track *Sequence:Repeating elements*, which contains about five million elements.

```
82749274 function calls (82747773 primitive calls) in 9066.371 seconds

ncalls cumtime percall filename:lineno(function)
...
 33167 8971.489  0.270 .../[1]:19(loadTrackView)
 33167 8934.596  0.269 .../[2]:7(start_and_end_indices)
 33167 8900.520  0.268 .../[2]:32(_get_region_start_and_end_indices)
 33167 8899.521  0.268 .../tables/table.py:1618(get_where_list)
 66334 8874.371  0.134 .../tables/table.py:1512(_where)
 33167 8853.943  0.267 .../tables/table.py:221(_table__where_indexed)
 66334 8759.313  0.132 .../tables/index.py:2016(get_chunkmap)
...
1: gtrackcore/track/pytables/TrackViewLoader.py
2: gtrackcore/track/pytables/database/IndexRetrievalQuery.py
```

Figure 6.2: Profile of index retrieval when querying the table with indexed 'start' and 'end' columns. Data from Appendix C on page 103, section 2.2.1.

The get_chuckmap method

From the profile, seen in Figure 6.2, we found that the method in PyTables called `get_chuckmap` seemed to be the bottleneck. One can see that the method is called exactly twice as many times as there are queries made to the table with the `get_where_list` method. As it turned out, the `get_chuckmap` method in PyTables is the one that retrieves the actual index of a column from disk, that will be used to efficiently find the rows that fulfill the query criteria for that column. While using 132 ms (milliseconds) to retrieve this index is not a very much time when querying a table once, it adds up to a lot when querying a table with two indexed columns 33 167 times.

We also tried to simply remove the indices, while still querying the table. We thus forced PyTables to do a sequential search, instead of a binary search, which is used by PyTables to search through indices. The overhead of retrieving the indices proved to be so much that the sequential search method decreased the total execution speed from 9066.4 seconds to 1730.9 seconds, using a total of 47 ms per query on average. As a reference, after the chunkmaps have been retrieved for the indexed table, a query seems to only takes about 2-3 ms on average. This number is based on the amount of time spent in the `get_where_list` method that is not spent on getting chunkmaps, and is to be concidered an estimate.

Binary search

Our solution to this problem was to not use table queries in PyTables at all, and rather implement our own binary search method, which was described in detail in section 5.7.2 on page 44. A profile of the *count of elements* operation with the new method is seen in Figure 6.3. The total time spent to perform the operation is reduced to 288.3 seconds, where the search for the start and end indices of the 33 167 track views took a total of 162.5 seconds, as opposed to 8934.6 seconds from the original method. This is an improvement that made the index search approximately 60 times faster. When performing the binary search, the time spent reading an element of a single row in the PyTables is about 0.1 ms. For this test a total of 454 207 rows are read, as seen in the profile from Figure 6.3.

Another interesting improvement is that the search for the end index uses almost half of the time as the search for the start index. This is because we use the result from the start index search as a lower bound restriction when searching for the end index, because the end index always is located after the start index.

The average time used to load a track view for the current PyTables solution is about 6 ms for the *count of elements* operation. As a comparison, the old memmap solution uses about 7 ms to load a track view for the same operation.

```
80688143 function calls (80687459 primitive calls) in 288.273 seconds

ncalls  cumtime  percall filename:lineno(function)
...
 33167  195.285   0.006 .../[1]:19(loadTrackView)
 33167  162.472   0.005 .../[2].py:12(start_and_end_indices)
 33167  128.925   0.004 .../[2]:37(_get_region_start_and_end_indices)
 33167   81.114   0.002 .../[2]:68(_start_index_for_segments)
 59592   79.235   0.001 .../[2]:113(_improve_min_and_max_index)
454207   69.053   0.000 .../tables/table.py:2077(__getitem__)
454207   59.681   0.000 .../tables/table.py:1904(read)
454207   47.870   0.000 .../tables/table.py:1832(_read)
 26425   46.960   0.002 .../[2]:95(_end_index_for_segments_and_points)
...
1: gtrackcore/track/pytables/TrackViewLoader.py
2: gtrackcore/track/pytables/database/IndexRetrievalQuery.py
```

Figure 6.3: Profile of index retrieval when using binary search to retrieve the correct indices. Data from Appendix C, section 2.2.1.

6.2.2 Open database connection

From section 5.8.1 on page 52 we know that the database connection is kept open until the Python process that initially opened a connection is terminated, and is closed using the `atexit` module. This is only happening when GTrackCore is *not* being used for preprocessing, i.e. when it is being used for data retrieval. Through profiling we found the opening of a PyTables file, i.e. a PyTables database connection, to be a very costly

operation because it was done very often. For every interaction with the database, the PyTables file was opened and then closed. When preprocessing a track, this is essential because the preprocessor switches between reading and writing from the connection. When retrieving data, the connection is only read from. Hence, the opening and closing of the connection was an unnecessary overhead, and consequently handled by keeping the connection open until termination. A possibly more time consuming operation was the retrieval of nodes, i.e. datasets or groups, from the PyTables file, as the `get_node` method is called many times.

From the profile in Figure 6.4, using the same *count of elements* operation as before, one can see the overhead of opening and closing the PyTables file for the 265 336 interactions with the file. Out of the total 2937.8 seconds spent on the operation, 1046.3 seconds is spent opening the file, and 202.4 seconds on closing it. An addition 1233.0 seconds are used to retrieve nodes from the PyTables file.

Figure 6.5 on the facing page shows the same profile, but with the optimization turned on, i.e. the file is kept open when in read mode. Out of the total 276.8 seconds the operation uses, only about 7.5 seconds are spent opening the database connection and retrieving nodes from it.

Caching nodes

Another small optimization regarding the somewhat costly node retrieval of PyTables has been node caching. All the calls in GTrackCore to the `get_node` method of PyTables is called from the `get_node` method in the database interface. From the profile where the connection is kept open, we see that the time spent on `get_node` in the `Database` object is much less than `tables.file.get_node`. This is because we cache the nodes in the `Database.get_node` method. It is important to notice that the `get_node` in PyTables is called multiple times from various places inside of PyTables itself, and that it is called recursively, being the reason for why it is called so many times compared to `Database.get_node`.

```
498290080 function calls (467046764 primitive calls) in 2937.774 seconds

 ncalls cumtime percall filename:lineno(function)
...
6122411 1233.314  0.002 .../tables/file.py:408(get_node)
 265336 1232.968  0.005 .../[1]:61(get_node)
 265336 1046.300  0.004 .../[1]:38(open)
 265336 1041.542  0.004 .../tables/file.py:222(open_file)
 265336 202.353   0.001 .../[1]:46(close)
 265336 201.281   0.001 .../tables/file.py:2707(close)
...
1: gtrackcore/track/pytables/database/Database.py
```

Figure 6.4: Profile of *count of elements* operation with a database connection that opens and closes for each interaction with the PyTables database. Data from Appendix C, section 2.2.1.

```
80688143 function calls (80687459 primitive calls) in 276.839 seconds

ncalls  cumtime  percall filename:lineno(function)
...
450899   6.634   0.000 .../tables/file.py:408(get_node)
265336   1.374   0.000 .../[1]:61(get_node)
     1   0.005   0.005 .../[1]:38(open)
     1   0.005   0.005 .../tables/file.py:222(open_file)
265336   0.237   0.000 .../[1]:147(close)
...
1: gtrackcore/track/pytables/database/Database.py
```

Figure 6.5: Profile of *count of elements* operation with a database connection to a PyTables file that is kept open until the Python interpreter terminates. Data from Appendix C, section 2.2.1.

Chapter 7

Performance results

This chapter provides performance results from benchmark testing, which are used to compare the memmap version of GTrackCore to the PyTables version, and also variances of the new version.

7.1 Overview

The tests that were used for comparing the old and new GTrackCore are the test operations presented in section 5.10 on page 53. A list of the operations are found in Table 7.2 on page 66, along with a short name for each operation that will be used later. A predefined set of tracks are chosen for each of the operations. Note that only the dataset with merged overlapping elements is being used in the tests. The following list describes the tracks, each with its track name, track type, number of columns, and number of elements in the track:

- **Overlap of two tracks**
  - Sequence:Repeating elements (hg19)
    Short name: Repeating elements
    Track type: Valued Segments. *Track elements:* 5 109 922
    Number of columns: 5
  - Chromatin:Roadmap Epigenomics:H3K27me3:ENCODE_wgEncodeBroadHistoneGm12878H3k27me3StdPk (hg19)
    Short name: Roadmap Epigenomics
    Track type: Valued Segments. *Track elements:* 23 331
    Number of columns: 8
- **Sum of weights**
  - DNA structure:Hi-C:Inter- and intrachromosomal:hESC:hESC-1M (hg19)
    Short name: Inter- and intrachromosomal
    Track type: Linked Genome Partition. *Track elements:* 3 053
    Number of columns: 5

Note: This track has weighted edges between all its elements, which means that both the edges and the weights columns have a dimension of 3 052, i.e. two lists of length 3 052 for each track element

- **Sum of values**

  - DNA structure:Bendability (hg18)
    Short name: Bendability
    Track type: Function. *Track elements:* 3 080 419 480
    Number of columns: 1

    Note: The datatype of the values for the track is a 64-bit floating point.

- **Count of elements**

  - Sequence:Repeating elements (hg19)
    Short name: Repeating elements
    Track type: Valued Segments. *Track elements:* 5 109 922
    Number of columns: 5

All of operations above are executable from the tool described in section 4.5 on page 27. The operations come with two configurable options: *data retrieval method* and *number of track views*.

Moreover, all the tracks used in the tests stores one bounding region for each of the chromosomes of the track, i.e. 24 bounding regions for each of the tracks.

For the *overlap* and *sum of weights* operations, one can select retrieval method to be either *NumPy array* or *iterator*. For *sum of values* and *count of elements* one can choose the number of track view by selecting which set of *genome regions* to use. There are two different sets of regions to choose from. The four operations, together with the configurations, total to eight different test cases that are meant to test various aspects of GTrackCore. These are explained in section 7.2 on page 66.

7.1.1 Number of track views

The two sets of genome regions are used to either load few or many track views using either the chromosomes or the genes of a genome, respectively. There are 24 chromosomes in the two genomes used (hg18 and hg19), which are used for creating few track view, and 28 336 genes for the *Bendabilty* track, and 33 167 for *Repeating elements*, both used for creating many track view.

In the tool, both the *sum of values* and *count of elements* operation can be run for either all the *chromosomes* or all the *genes* of the genome, while *overlap* and *sum of weights* is only for the whole genome, i.e. all the chromosomes.

Genes

The genes of a genome are described by the genomic track *Genes and gene subsets:Genes:Ensembl*. When the genes are used as regions for a tool, they are read from a textual genomic track, and converted into *genome regions* that are used by the tool to load track views. The overhead of this conversion is about 15 seconds in total for the *sum of values* operation with genes, and around 70 seconds for *count of elements* operation. The conversion is done using the `UserBinSource` class in GTrackCore, where all the elements also are sorted.

The average length of each genome region created from the genes, from both genome hg18 and hg19, is about 40 000-50 000, but range from lengths of a few base-pairs to a couple million base-pairs. Hence, using genes as track view not only lets us load many of them, but also in a large variety of lengths.

7.1.2 Retrieval method

In GTrackCore one retrieves data either as *NumPy arrays* or by using an *iterator*. For most test cases, the NumPy array retrieval method is used. Note that both of the iterator-based operations (see section 7.2 on the next page) for the PyTables version always retrieve data from a PyTables table, and never from PyTables arrays. Moreover, both of the *count of elements* operations do not read any actual data from the track view. For this reason, the retrieval method is set to 'N/A' (not applicable) in the tables in the section 7.2 on the following page. For the same reason, the *count* operations are not used when comparing table columns and arrays of PyTables.

7.1.3 File size of the test tracks

Table 7.1: The table shows the file size of the test tracks for the PyTables files, one for each of the tested compression levels. The sizes are in megabytes.

	Compression level			
Track	**0**	**1**	**5**	**9**
Bendability	49 290.06	46 053.66	41 545.36	38 287.06
Roadmap Epigenomics	16.33	7.36	3.10	3.03
Inter- and intrachromosomal	392.06	314.71	169.87	160.71
Repeating elements	1 693.49	557.20	276.37	275.65

For each of the compression levels that GTrackCore is tested with, when using PyTables, there is a separate file that data is retrieved from. Table 7.1 shows the file sizes for each compression level of the tracks used by the different operations, which were described above.

It is important to notice that all of these files includes the track stored both as a table and as arrays. The difference in size for a PyTables file with only a table, and one with both a table and arrays, is almost exactly twice as large from the former to the latter. The memmap files for the old version are not in any way compressed, and use the same amount of space as an uncompressed PyTables file when the file only contain either a table *or* arrays.

7.2 The test cases

Table 7.2 shows a complete overview of all the test cases that are used to compare the GTrackCore versions. For each of the test cases, a *short name* is given for later usage, most with a letter at the end referring to a configuration. The (I) means 'iterator', (C) is 'chromosomes', and (G) refers to 'genes'. It is implicitly defined that both *overlap* and *sum of weights* are operations that use the chromosomes as track views, and that all operations without '(I)' uses NumPy array as retrieval method.

Table 7.2: The table shows the complete set of test cases that can be performed by the operation performance tool, with the two different configurations for *number of track views* and *retrieval method*.

Operation	**Number of track views**	**Retrieval method**	**Short name**
Overlap	24 (Chromosomes)	NumPy array	Overlap
		Iterator	Overlap (I)
Sum of weights		NumPy array	Weights
		Iterator	Weights (I)
Sum of values	24 (Chromosomes)	NumPy array	Values (C)
	28 336 (Genes)		Values (G)
Count of elements	24 (Chromosomes)	N/A	Count (C)
	33 167 (Genes)		Count (G)

7.2.1 Test case description

The following list describes what aspect of GTrackCore each case is meant to cover. Each case tries to be as enclosing as possible, in the meaning that only one functionality is supposed to be tested for each case. This is, however, difficult, as a single operation usually uses multiple functionalities of GTrackCore.

Overlap The 'Overlap' test case is meant to test the use of a common operation for genomic tracks, for two Segments tracks, where one of them is small (23 331 elements), and the other is medium sized (5 109 922 elements). Moreover, this case tests the retrieval speed of medium sized tracks.

Overlap (I) This case tests the performance of the track view iterator, using the same tracks as in the previous case. In this test case it is important to notice the difference between how the track view iterator works in the old and new GTrackCore version, which is described in section 5.6.2 on page 39.

Weights The 'Weights' operation is meant to test the retrieval of NumPy array data with a large shape. The *Inter- and intrachromosomal* track has weighted edges between all of the 3 053 track elements, meaning that the each track element has weights to 3 052 elements. When retrieving the weights as a NumPy array, the shape is `(<num_elements>, 3052)`, where `<num_elements>` is the number of elements in the track view one retrieves the NumPy array from.

Weights (I) The iterator version of the 'Weights' operation uses the graph view interface to retrieve all the weights of the track. The graph view edge iterator loops through all the edges of every track element. Because the edges refers to track elements at any position in the track data, this operation will test the random access performance of the underlying data model.

Values (C) The *Bendability* track contains over three billion 64-bit floating point values, separated into 24 chromosomes. The 'Values (C)' case tests the retrieval of large NumPy arrays, where the largest contain about 250 million elements, or about 2 GB of data.

Values (G) The 'Values (G)' test case operates on the same track as the previous one, but here, many more and much smaller NumPy arrays are retrieved. This operation tests loading many track view *and* retrieving many small to medium sized subsets of data from the track.

Remember that because the *Bendability* track is of type Function, the method for finding the start and end indices in the track view loader is a simple calculation, and is very fast.

Count (C) As described in section 5.6.1 on page 39, the counting of elements is a task performed for every initialization of track views. Because counting of elements is already being done, we use the *count of elements* operations as a method of testing the *track view loader* independently. For 'Count (C)', there are 24 track view loadings, using chromosomes as genome regions.

The track chosen for this operation is *Repeating elements* because it contains over five million elements, thus making task of finding indices of the track view somewhat non-trivial, as opposed to using a track with very few elements.

Count (G) Like with 'Count (C)', this case tests the *track view loader*. The difference is that 'Count (G)' uses the genes of the genome as genome regions, hence having 33 167 track views loaded.

Basic operations

The test cases above each tries, as mentioned, to be as enclosing as possible. These individual tasks can be viewed as the *basic operations* of the data retrieval part of GTrackCore. These basic operations are defined to be: reading contiguous data, iterating track elements of a track view, loading track views, and random data access.

7.3 Results

This section contains all the results from benchmark tests of GTrackCore, both for the old memmap version and the new PyTables version. The first section provides the raw data from running the test cases for the memmap solution and all the variations of PyTables. The next sections present the data more clearly to highlight differences. Firstly, a simple comparison between the two GTrackCore versions, and secondly, graphically presentations of the variances of the PyTables solution.

When presenting some of the data in this section, results from detailed profiling will be used to emphasize the root of the benchmark result, i.e. what that actually is taking time. The results from the profiles includes, however, an overhead, making them less accurate than the benchmarks. When comparing two results, this will not have any impact, other than a relatively small increase in time for the operation being compared, for both GTrackCore versions.

7.3.1 Raw data

The raw data from running the benchmark tests are presented in Table 7.3a and 7.3b on the next page. The original data can be found as supplementary material in Appendix C on page 103.

7.3.2 Memmap compared to PyTables

To depict the overall differences between the old and new GTrackCore solution, the two versions are compared directly against each other, using the average running time for each of the test cases as a metric. While the PyTables version contains many variations, not all of these are used in this comparison. Here, the retrieval method for NumPy arrays is *table columns* and the dataset being used is uncompressed. We will in section 7.3.3 on page 71 compare variances of PyTables.

Table 7.4 on page 70 shows a list of the running times for each test cases for both GTrackCore versions. Moreover, it shows the change in time, in percentage, *from* the memmap version *to* the PyTables version. To illustrate the difference, Figure 7.1 on page 71 shows a bar chart of the same data, where the two versions of GTrackCore are put up against each other for each test case.

Table 7.3: Average of 10 runs for all test cases for both the memmap and PyTables versions of GTrackCore. All the results are in seconds.

(a) Average of 10 runs for all test cases with memmap.

Operation	**Retrieval method**	**memmap**
Overlap (I)	Iterator	677.5169
Weights (I)		99.2393
Overlap	NumPy arrays	1.7083
Weights		0.3456
Values (C)		29.1901
Values (G)		115.6236
Count (C)	N/A	0.1352
Count (G)		285.8031

(b) Average of 10 runs for all the test cases with all the variations of PyTables. This includes retrieval of data from

		Compression level			
Operation	**Retrieval method**	**0**	**1**	**5**	**9**
Overlap (I)	Iterator	75.6562	76.9853	78.057	77.0294
Weights (I)		95.7631	96.584	98.1806	95.4432
Overlap	NumPy array Table columns	3.3547	4.7659	4.8027	5.8802
Weights		0.5894	0.5997	1.2514	1.2629
Values (C)		53.3525	56.3878	79.3238	126.2875
Values (G)		96.7433	120.1512	121.4629	151.339
Overlap	NumPy array Arrays	2.3142	1.9776	2.0393	1.9135
Weights		0.6094	0.573	0.9737	0.9821
Values (C)		45.3136	50.9591	63.4918	87.6784
Values (G)		83.6112	83.0448	91.6294	107.4889
Count (C)	N/A	0.2477	0.3372	0.413	0.4698
Count (G)		231.9679	337.3673	340.8068	406.2412

Table 7.4: The table shows a comparison of memmap and PyTables for all the test cases, with a percentage change *from* the memmap *to* the PyTables version. For the PyTables column, the NumPy array source is *table columns*, and the datasets are uncompressed (Compression level 0). All the results are average of 10 runs, and are taken from Table 7.3a and 7.3b on the previous page.

Operation	**Retrieval method**	**memmap**	**PyTables**	**% change**
Overlap (I)	Iterator	677.5169	75.6562	-88.83%
Weights (I)		99.2393	95.7631	-3.50%
Overlap	NumPy arrays	1.7083	3.3547	96.38%
Weights		0.3456	0.5894	70.54%
Values (C)		29.1901	53.3525	82.78%
Values (G)		115.6236	96.7433	-16.33%
Count (C)	N/A	0.1352	0.2477	83.21%
Count (G)		285.8031	231.9679	-18.84%

Results

The following paragraphs sums up the results from the comparison between the memmap and PyTables solution of GTrackCore, and is separated into categories of which basic operation that is being tested.

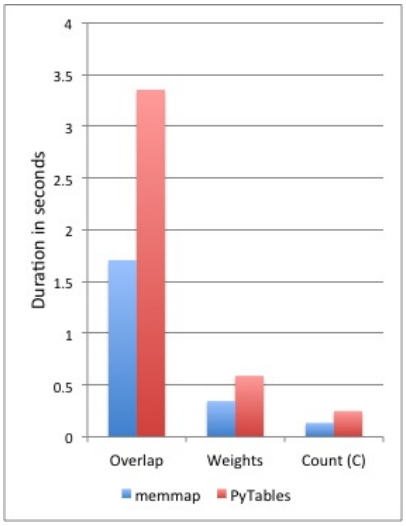
Track view iterator From the 'Overlap (I)' test case, we see that PyTables is almost 9 times faster than when using memmaps. As described before, the PyTables version takes advantage of a highly optimized iterator, while the memmap solution uses an incrementing index referring to a posistion in the track data (the `SmartMemmap` objects).

Looking at the profiles of the operation for both versions, we that the memmap solution uses a total of 644.715 seconds on reading data, while the PyTables version spends 62.038 seconds on the same task.

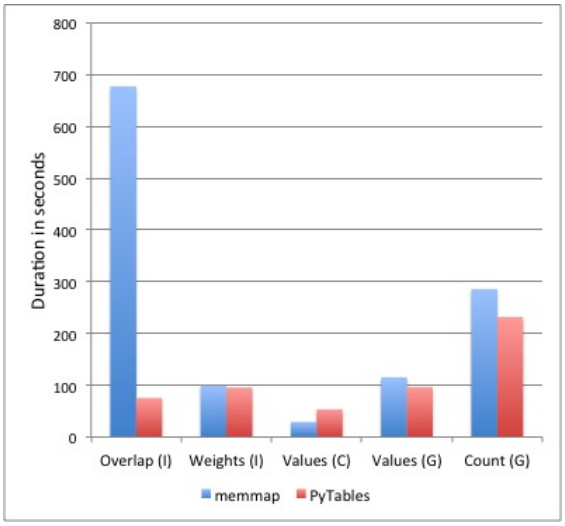
Reading contiguous data From both 'Values (C)', 'Overlap' and 'Weights', the performance results favors the memmap solution, even if the slices of data being read are large, medium, or has a large shape. The difference is close to twice as fast for the memmap version.

Loading track views When it comes to loading track views, the PyTables solution has a slight advantage, as seen in 'Values (G)' and 'Count (G)'. For the former, the decrease in loading time actually evens out the time spent reading data, making the PyTables version faster in total.

Since the track being used for 'Values (G)' is a Function track, the part of the track view loading that is spent on finding the start and end indices is only a simple calculation, which is described in section 5.7 on page 43.



(a) The operations where the running time is relatively low.



(b) The operations where the running time is relativly high.

Figure 7.1: The figures show a bar chart with the difference in average running time between the memmap version and the PyTables version of GTrackCore for each of the test cases. The figure is split into two charts simply because of the different time scales for the operations. The chart to the left shows the differences for the relativly fast operations, while the one on the right is for the relativly slow operations. The data is taken from Table 7.4 on the preceding page

The average time spent for each track view load is for the PyTables version approximately 1.2 ms, and a bit over 2.5 ms for the memmap version.

Looking at profiles of the 'Count (G)' test case, where the track is a Segments track, and a search for the correct indices is required, we see that the average time spent loading track views for the memmap solution is 6.5 ms, while it is about 5.2 ms for the PyTables version.

For 'Count (C)', the loading takes on average 6.6 ms for the new version, and 4.5 ms for the old one, being almost the opposite result of when having many track view loads.

Random data access The two versions of GTrackCore perform equally in the 'Weights (I)' test case, which test accessing data randomly in the track data, with a slight advantage to the PyTables version. We will discuss this further in Chapter 8.

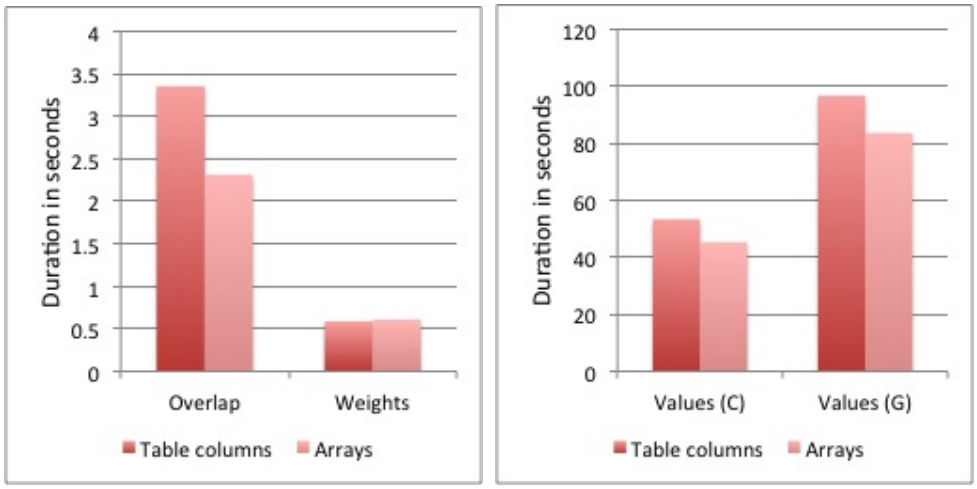
7.3.3 Variants of the PyTables version

As mentioned before, the PyTables version is tested with two different configurations. The first being retrieval method as either *table columns* or *arrays*, and the second being four different compression levels. These configurations will be presented separately. Firstly, the results from the two different retrieval methods, and secondly, the results from using different

compression levels.

Table columns vs. arrays

As mentioned, both the iterator-based operations and the count operations do not retrieve data from arrays, and is accordingly not used in this comparison. The differences in the the retrieval methods is illustrated in Figure 7.2.



(a) The operations where the running time is relativly low.

(b) The operations where the running time is relativly high.

Figure 7.2: The figures show a bar chart with the difference in average running time between using the table columns or arrays as data source for retrieval. The figure is split into two charts simply because of the different time scales for the operations. The chart to the left shows the differences for the relativly fast operations, while the one on the right is for the relativly slow operations. Data is taken from Table 7.3b on page 69

Reading contiguous data From the figure, we see that three out four test cases show that using arrays is faster than using table columns. I.e. all test cases except 'Weights'. To get a deeper look at the operations, we will look at the performance of the `as_numpy_array` method in the `VirtualTrackColumn` class, which is the method where data actually is being read from either a table column or an array. Table 7.5 on the facing page shows the average reading time per read for each of the test cases. Here, we see that reading data from arrays always is faster than reading data from table columns. While Figure 7.2 shows the complete running time for each test case, Table 7.5 shows how much time that was spent *reading data* for each test case, and does not include any other aspect of the operations, such as track view loading and NumPy operations. The following list sums up the results from the table:

- 'Overlap' shows that with a medium sized track it is about 20 times faster to do few (24) reads from arrays than from table columns

- From both 'Weights' and 'Values (G)' we see that reading from arrays is about twice as fast than table columns when reading a small number of elements from the dataset
- When reading very large segments, the average reading time is about 10 % faster from arrays than from table columns

Table 7.5: Comparison of data reading with table columns and arrays in PyTables. The table shows the average reading time per read for each test case. All values are in milliseconds (ms). The data is from the raw profiling data found in Appendix C.

Operation	**Table columns**	**Arrays**
Overlap	11 ms	0.5 ms
Weights	8.5 ms	4.5 ms
Values (C)	1 348 ms	1 203 ms
Values (G)	1 ms	0.5 ms

Performance with different compression levels

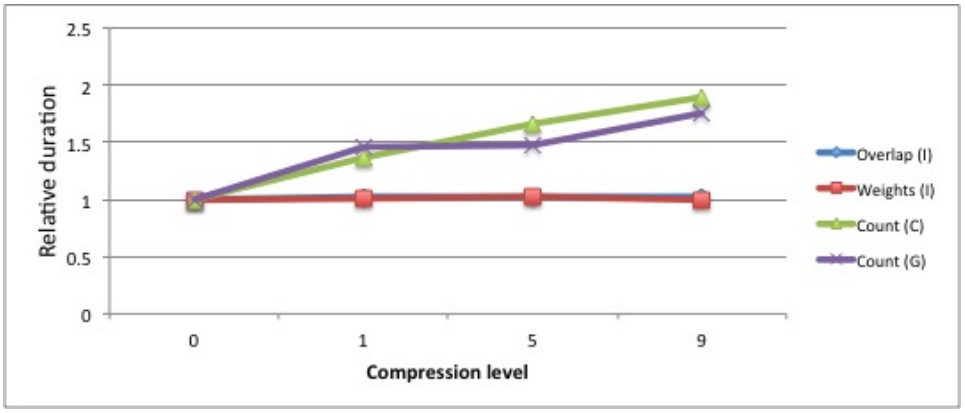
This section presents the results from each test case of the PyTables version, comparing the performance with different compression levels on the datasets. The four different compression levels being tested are 0 (no compression), 1 (low compression), 5 (medium compression), and 9 (high compression), all using the Blosc compression filter.

Figure 7.3 on page 75 shows how each of the tested compression levels affect the retrieval speed for each of the test cases. The results are illustrated by using the uncompressed dataset as a reference, and then comparing the other compression levels, i.e. 1, 5 and 9, to that. In other words, the figure shows the change in performance for each compression level as a factor between the uncompressed data and the current compression level.

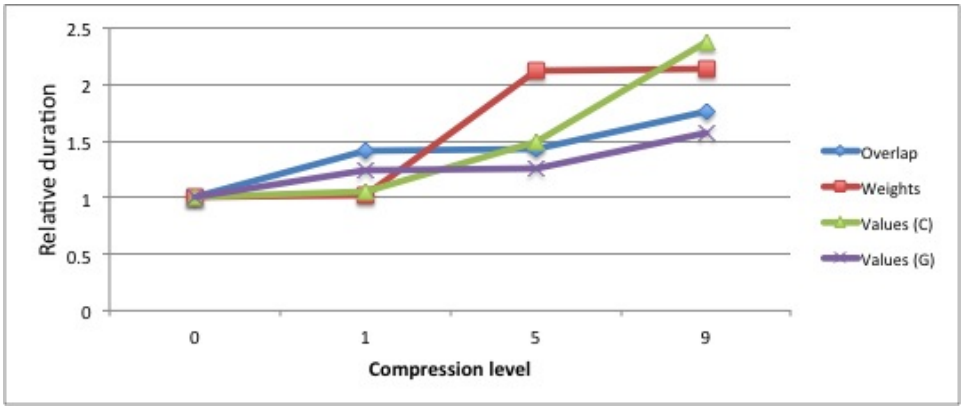
Compression results From the graphs in Figure 7.3 on page 75, we see that for nearly all test cases, compression has either a negative or indifferent impact on performance. It is only for 'Overlap' with arrays that has a positive development on performance as the compression level increase. Here, the running time for compression level 9 is about 80 % of the initial time. Moreover, the iterator-based operations are practically unaffected by compression. For the most of the remaining test cases, for compression level 9, the results show that compression increased time usage by a factor from 1.5 to 2.

However, for compression level 1, the overall increase was not as much. Here, all test cases performed with an increase in time less than 50 %, compared to the original time. When using *arrays* as retrieval method, it was

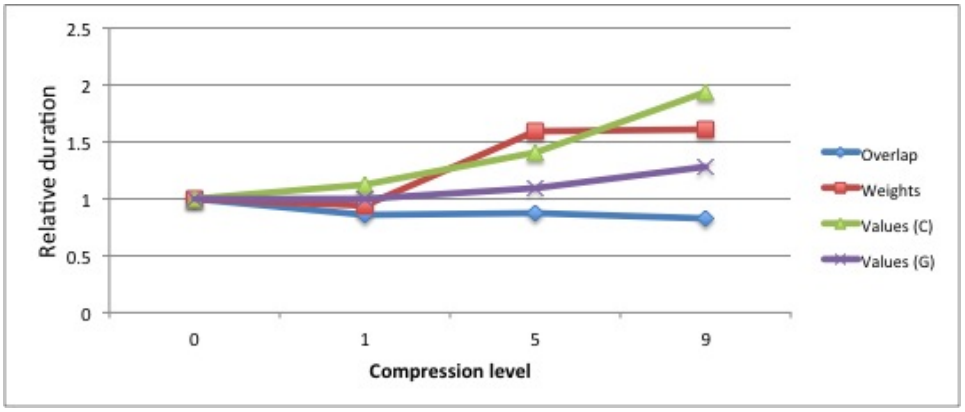
only the 'Values (C)' test case that performed worse than when retrieving uncompressed data. The other three tests showed a minor decrease in time consumption. For *table columns*, the results show that 'Overlap' and 'Values (G)' slows down by a factor of 1.42 and 1.24, respectively, while 'Weights' and 'Values (C)' remain virtually unchanged.



(a) Relative performance for the iterator-based test cases, along with the two 'Count' test cases.



(b) Relative performance of the test cases with NumPy array as retrieval method, using *table columns* as data source.



(c) Relative performance of the test cases with NumPy array as retrieval method, using *arrays* as data source.

Figure 7.3: The figures show the change in performance for each compression level compared to the performance when using uncompressed data. Each test case has the value *1* for compression level 0, i.e. no compression. The subsequent values along the horizontal axis are all a relative change in time to the initial value, one marked line for each of the test cases. The data is taken from Table 7.3b on page 69.

Chapter 8

Discussion

In this chapter we discuss the proposed new GTrackCore solution against the old one based on results from the previous chapter, as well as issues and possible alterations to the solution.

8.1 Overview

The main purpose of this thesis is the investigation of whether the PyTables library is a suitable alternative as a new data model in GTrackCore, as opposed to using the existing memmap solution. To answer this, we first need to define what suitable means. In this work, regarding data retrieval, we define PyTables to be suitable if the following criteria are met:

- PyTables include the functionality to not limit any functionality of GTrackCore
- Using PyTables will not decrease the efficiency of data retrieval in GTrackCore significantly

Furthermore, we wanted to see how one can utilize some of the advantages of the PyTables library in GTrackCore, such as querying indexed tables for efficient data retrieval.

We will first discuss the actual results from the previous chapter, and how data retrieval in GTrackCore has been affected in different areas when using PyTables and HDF5 as a data model, rather than memmaps. Afterwards, we will discuss the research questions of this work more directly.

8.2 Memmaps compared to PyTables

From the results in the previous chapter, we learn that the performance of both solutions are within the same order of magnitude in most settings, i.e. relatively equal. The overall result shows that it is generally faster to read data from memmaps than from an HDF5 file using PyTables. However, the performance of the track view iterator in the PyTables version of GTrackCore is one result that stands out as special. The 'Overlap (I)' test case showed that using the row iterator of tables in PyTables is almost

10 times as fast as the method used in the memmap version. We will come back to this in the next section.

The profiles referred to in this section is seen in Appendix C, section 2.1.1 for the memmaps solution and 2.1.2 for the PyTables solution.

8.2.1 Comparing basic operations

By discussing each of the basic operations separately, the differences of using NumPy memmaps and the PyTables library and HDF5 in GTrackCore is efficiently compared and demonstrated.

Reading contiguous data

As mentioned in the beginning of this chapter, the results showed an advantage for the memmap version when it comes to reading segments of contiguous data from disk, regardless of the size and shape of the data. This is likely due to the fact that memmaps are stored contiguously on disk, while HDF5 stores data separately on disk in chunks. Having the data stored as chunks provides, for instance, the possibility to compress individual chunks of data, but also gives additional overhead when reading it. We discuss this further in section 8.3.1 on page 81.

The track view in GTrackCore relies on using NumPy ndarray objects. The NumPy memmap is a direct subclass of the ndarray class. Moreover, a memmap file is simply a direct binary file representation of an ndarray. Reading memmaps into memory, and using these as NumPy arrays is therefore fairly straightforward, and hence, very fast. PyTables, however, stores data in an HDF5 file in chunks of data. When reading slices of contiguous data in PyTables, one have to retrieve the requested slices into memory from possibly multiple chunks and apply any potential filters to each chunk. To know where the chunks are located on disk, HDF5 has a map of the chunks, with pointer to each of them. For each of the chunks read, there is a single I/O read operation. The HDF5 library do, however, include a chunk cache. If the same data is read twice, or if two different subsets of data is in the same chunk, it may be very quickly accessed the second time.

'Values (G)' The 'Values (G)' test case seems to prove that the PyTables version is faster when reading smaller pieces of data. Looking at the profiles for these operations, we see that `loadTrackView` method stands for a total of 72 % of the total running time for the memmap version, and only 40 % for the PyTables version. The total time spent reading was about 8 % of the total running time for the old version of GTrackCore, and 28 % for the new. This means that the reading still is faster in the memmap version.

Track view iterator

As the results proved, the track view iterator is almost 10 times faster for the PyTables version than the memmap version. The row iterator of tables

in PyTables is written in Cython, and hence, very fast. Furthermore, it is a highly optimized iterator, as it is used for multiple purposes in PyTables.

As we remember from before, the method for iterating the track elements of a track view in the memmap version is to use an index variable, which is referring to a position in the memmap files. This is incremented by one for each call to the `next` method. When retrieving data from a field of a track, the value at the indexed position is retrieved from memmap for that given field.

Retrieving single values from the track elements yielded by the track view iterator is about 10 times faster for PyTables than memmaps. The PyTables iterator uses 1.4 μs (microseconds) for each retrieval on average, while the same task takes 15.9 μs for memmaps. As all aspects of the iterator is equal in both GTrackCore versions, except for the underlying data structure, this increase in speed is probably a consequence of the HDF5 chunk caching. When a chunk of data is read, it is put in memory. This way retrieval of data from the same chunk is very fast. Since we iterate the rows in the order they appear on disk, we take full advantage of the caching as consecutive rows almost always are located in the same chunk. While NumPy memmap probably have some read-a-head and caching, it is not as effective as the chunk caching in HDF5. In addition, one have to read from different files when accessing multiple field when using memmaps.

Loading track views

Nearly all of the time elapsed when loading track views for tracks with either *starts* or *ends* attributes, is spent finding the correct indices in the track data according the the requested genome region. This includes both finding an enclosing bounding region, and within that region, the actual indices. From the 'Count (G)', we see that the PyTables version, using a binary search algorithm to find find the indices, is slightly faster than the memmap version. Here, the left- and rightIndex, described in 5.7.1, are used to efficiently find the indices within the bounding region. The profiles for the 'Count (G)' operation show that the average track view load for the PyTables version is 5.5 ms, while it is 7 ms for the memmap version.

Using the left- and right indices to find the indices of the track view is actually a bit faster than the binary search of the PyTables version. However, that method do not find the exact indices. Those are found later by performing a sequential search, which took about 2.3 ms on average for 'Count (G)'. The 'Count (C)' was a bit faster for the memmap solution. This was mostly because the sequential search went very fast when the start index of the track view was the very first element in the search. Here, it only took 0.5 ms on average for each of the 24 track view loads.

The memmap version From the profile, we see that the memmap version spend 2.9 ms on the index lookup in the track view loader. This number also includes other overhead in the track view loader, such as object creations. An additional 2.3 ms is spent refining those indices to be exact, using the sequential search. Furthermore, 1.8 ms is spent finding the bounding region.

Much of this time is spent opening a Python shelve, where the bounding regions are stored. This process do however include caching to reduce the number of file openings.

To reduce the time of the sequential search, one can suggest using a right- and left index that has higher resolution than the one being used today, which indexes every 100 000th base-pair. If one, for instance, used an index that was twice as large, i.e. indexed for every 50 000th base-pair, the sequential search time would, on average, be cut in half. The size of the indices would then be twice as large.

The PyTables version By analyzing the profiles, we see that for the PyTables version of the track view loader, 3.5 ms is spent on the binary search, which includes a small sequential search, and about 0.5 ms on finding a bounding region. An additional 1.5 ms on average is other overhead in the track view loader.

The task of finding the bounding region is, as mentioned, a query on the bounding region table. The table is not indexed (see section 8.6 on page 87), so the query will therefore have a linear growth in time consumption for tracks with more bounding regions. The time spent for the binary search is, however, fairly stable. For the 'Count (C)' operation, the time spent is 4.7 ms. The reason for why it takes about 1 ms more time here is probably because the binary search uses a maximum number of steps in the search when the result is at the lower and upper limit of the search.

Random data access

The 'Weights (I)' test case was almost equal for both versions. The PyTables version is only about 3.5 % faster. Much of the total time for this operation is overhead that are equal in both versions, such as creating many objects, and hence, do not test the actual underlying data model. From the source code, we see that the actual reading of weights, and edges, is happening in the method `getNeighborIter` in the `NodeElement`[1] class, which represents a track elements in a graph view. From the profiles of the operations, we see that this method is called 9 323 862 times. The average time used for the PyTables version for this method is about 9 μs, and 8.5 μs for the memmap version.

From Table 7.1 on page 65, we know that the track used in the 'Weights (I)' test case only is about 200 MB of data, which easily can be held in memory. As mentioned, each track element in the track has a weighted edge to all the other track elements in the track. The way the graph view iterator works is that all the edges and weights of a track element is read for every call to `getNeighborIter`. However, because of caching, the data is actually only read from disk once for each track element, i.e. only a little over 3 000 disk reads.

[1] Called PyTablesNodeElement in the PyTables version of GTrackCore

Inconclusive result Because of how the graph view iterator works, the small size of the track, and simply because of caching in both GTrackCore version, and probably also disk cache, this test case do not show a significant advantage in either of the version. While it seems that the memmap version is slightly faster, this test is inadequate to conclude on any performance advantages regarding random data access time. This inconclusiveness is also the case for the 'Weights' test case, where reading contiguous data of large shape was supposed to be tested.

8.3 Variants of PyTables version

In this section, we discuss the results from using arrays as opposed to table columns in the 'as NumPy array' methods, and what effect compression has on data retrieval in PyTables. Additionally, we look deeper into what impact different chunkshapes on the HDF5 dataset has on data retrieval. As this is not thoroughly tested, we will look at some experiments.

8.3.1 Table columns and arrays

The results from the 'as NumPy array' tests show that reading contiguous data from arrays always is faster than reading from table columns. There are mainly two reasons for this behavior, where the second is a consequence of the first. The first reason is that tables in PyTables are stored as a compound datatypes in HDF5, while arrays are stored as atomic datatypes. The second is that the automatic chunkshape set by PyTables will cover fewer track elements for tables than for arrays.

Compound datatype

As described in section 2.3.4 on page 13, the compound datatype in the HDF5 format is very similar to a struct in C. When creating a table in PyTables, a dataset is created in the HDF5 file with a compound datatype. This datatype has the same fields as the *table description* that was used to describe each column of the table in PyTables. When rows are created in PyTables, objects of the compound datatype are made. These are then appended to the dataset. This means that each row of a table is physically stored as a single unit on disk, and that the rows are stored consecutively. This type of ordering is known as *row-major order*[1].

The figure titled 'Dataset with compound datatype' in Figure 8.1 on page 84 shows an example of how tables in PyTables are stored in an HDF5 file. In the example we see a table with three fields: an integer (2 bytes), a floating-point number (4 bytes), and a string (12 bytes). The dataset has a chunkshape that is chosen based on having chunks of 36 bytes in size. This is not exactly how PyTables determines the chunkshape when using automatic chunkshape, but in principle it could as simple as having a max byte size for each chunk. The table contains 4 rows, each containing 18 bytes of data.

[1] http://en.wikipedia.org/wiki/Row-major_order

Hence, the chunkshape is set to 2. This means that there are a maximum of 2 data elements in each chunk. Since the dataset contains 4 rows, there are a total of 2 chunks.

If we want to retrieve all the integers elements, i.e. the blue elements, in the dataset, e.g. by using an 'as NumPy array' method in GTrackCore, we have to also retrieve all the other values for each row object. This is because HDF5 treats each chunk as an atomic datatype when performing I/O. It is not possible to only read parts of chunks or compound datatypes. Afterwards, we can select the 2 bytes from each object that represents the integer number. Since the data is separated into 2 chunks, there will be 2 disk reads in this example.

Atomic datatype

The figure titled 'Datasets with atomic datatypes' in Figure 8.1 on page 84 shows the exact same data as before, with the exception that it is split into multiple datasets, where each has an atomic datatype. These datasets represents a single array in PyTables, and, e.g. a single column in a genomic track. As we can see, the chunkshapes in these datasets are also set based on having chunks of 36 bytes. Here, however, each atomic datatype takes less space than the compound datatype, which gives a larger chunkshape. Hence, we can have more elements in each chunk. Each of the datasets with atomic datatype has a length equal to the previous compound dataset, i.e. the length 4. In the drawing, each element is separated by a slightly thicker vertical line.

If we here want to retrieve all the elements in the array with integers, by using an 'as NumPy array' method, we simply retrieve them in a single disk read. It is this difference in storage method between tables and arrays and that makes retrieving data from array faster than from table columns.

Example of difference between tables and arrays

Let us say that the datasets used in the figure contained 1 000 elements instead of 4, and that we still want to retrieve all the integers. When using tables in PyTables, this would give 18 bytes $*$ 1 000=18 000 bytes of data, separated into 18 000 bytes/36 bytes=500 chunks. This means that reading the integers would involve reading 18 kilobytes of data in 500 disk reads. When using arrays in PyTables, we would get 2 bytes $*$ 1 000=2 000 bytes of data, separated into 2 000 bytes/36 bytes $\approx$ 56 chunks. Here, we reduce the total number of reads from disk, and the number of bytes read, by a factor of almost 10.

Analysis of results

From the results in the previous chapter, we saw that the average reading time of each read operation was faster for arrays. However, the increase in speed was not constant in all the test cases.

For both of the 'Values' test cases we use the track *Bendability* to read values from. This track contains only one column. This means that both

the table and the array contain the exact same data, and has the same chunkshape. Based on what we have discussed in the previous sections, these should perform equally. However, the array is still stored as an atomic datatype, and the table is stored as a compound one. It is probably because of some extra overhead with using compound types and type-checking that makes slicing a table column a bit slower than an array.

The 'Overlap' test case had a reading time that was 20 times faster with arrays, as opposed to tables columns. This is likely due to the fact that the chunkshape of arrays used by the operation (the *start* and *end* columns) is nearly 20 times larger than the chunkshape of the table used for the same operation. This means that 20 times more data that actually will be used by the operation is read for each disk I/O.

As mentioned earlier, the track used in the 'Weights' test case was too small to give any conclusive results. Nonetheless, in theory, the smaller chunkshape should result in a faster reading when the data has large shapes as well.

Using arrays in other settings

Because using PyTables arrays only was tested for the 'as NumPy array' methods, we do not know how arrays will perform in other settings, e.g. in the track view iterator and the track view loader.

Track view iterator Using the row iterator in the tables proved to be a very efficient method of iterating the track elements of a track view, as described before. Using arrays instead would very likely decrease performance. This is because arrays in PyTables do not include an optimized iterator, and because one would have to read from multiple datasets simultaneously, resulting in more disk I/O.

Even so, investigating the performance of arrays in the track view iterator is a path worth pursuing, as it may prove to be faster than using memmaps, while probably slower than using tables.

Track view loader As mentioned, we have not properly tested arrays in all setting where we have tested tables. This also counts for the track view loader. It is, however, very possible that the binary search algorithm used in the PyTables version of the track view loader would be faster when using arrays. This is because when retrieving data from tables, one has to read data from disk that one might not need, i.e. other columns of the genomic track than one wants. When using arrays, one will only read data associated with a single column, e.g. the *starts* column.

8.3.2 Compression level

Why we wanted to test the impact of having compressed data in a data retrieval setting is not immediately clear, as decompressing the data for every retrieval only seem to provide more overhead. The reason, however, is that the documentation of PyTables made it clear that it may, in some

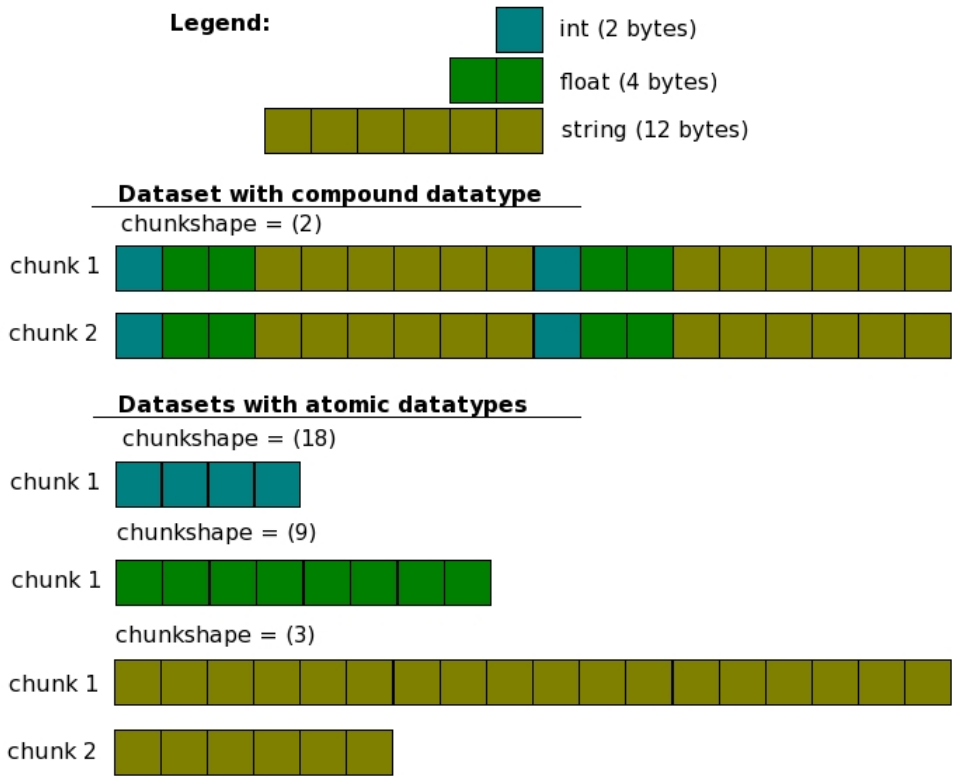


Figure 8.1: The figure shows an example of chunks in HDF5. To illustrate the difference between tables and arrays in PyTables, the figure shows two representations of the same data. One where a compound datatype in HDF5 is used on a single dataset (a table), and one where three datasets with atomic datatypes are used to store the same data (arrays). Moreover, a chunkshape is set for the datasets based on having chunks of 36 bytes.
To simplify the figure and the example, each dataset only contain a single dimension. The principle is, however, the same with any number of dimensions.

cases, decrease the total read time. This is because compressed data takes less space on disk, and hence, takes shorter time to read from disk. This decreased time spent reading data may compensate for the time it takes to decompress the data, so that the overall time spent on a read operation is less than when using uncompressed data.

We chose to use the Blosc compression filter because it is custom made for PyTables, and designed to decompress quickly. Moreover, we chose to test four different levels of compression as different usage scenarios may be better suited for different compression levels.

The results showed, however, that compression rarely provided any performance improvements. The time it takes to decompress the data takes more time than is saved by reading less from disk. For arrays, compression had less of a negative impact than when retrieving data from table columns. This is probably because the number of disk reads decreased more for arrays, than for tables, as we do not need all the columns of a track.

The 'Overlap' test case, when using arrays, was the only one that decreased the total running time as the compression level raised. This seems to prove that compression still can be have a positive effect of data retrieval, as suggested by the PyTables documentation.

Iterator test cases As seen in the results, both the track view and the graph view iterator was nearly unaffected by increasing the compression level. For the graph view iterator, this is likely due to the small size of the track, as mentioned before.

For the track view iterator, the steady performance speed is probably a consequence of chunk caching in HDF5. Chunks that are read from disk are cached *after* they have been decompressed. Therefore, the total overhead of having compressed data becomes insignificant in the total operation. The profiles (Appendix C, section 2.2.3) also show that there are no remarkable changes between each compression level.

Compressed data

While compressing data do decrease performance of nearly all test cases, the disk space used for each track is reduced, some times even substantially, as shown in Table 7.1 on page 65 and more detailed by Rongved in [34]. If GTrackCore were to be used in a setting where disk space is an issue, e.g. on home computers, reducing the file size of the genomic tracks can be helpful, even if track operations become somewhat slower.

8.3.3 Impact of chunkshape of a dataset

When creating datasets in PyTables, i.e. tables or arrays, one can either specify a chunkshape explicitly, or let PyTables calculate a sensible shape for you automatically. The PyTables documentation recommends using the latter, which is what we did in the implementation. However, the optimization guide provided by PyTables ([29]) states that one should experiment with chunkshapes to find what works best for your application. This is something that not has been thoroughly tested by us. We have, however, a short experiment with two different chunkshapes in Appendix D on page 105. One of them has the automatic shape calculated by PyTables, while the other is a little over 100 times larger than the first.

In the experiment we see that large slices of data is faster to retrieve when the chunkshape is large, and small slices it is faster with a smaller chunkshape. When one has a large shape, but only want to retrieve a small subset of the data, one still needs to retrieve the whole chunk as HDF5 only performs I/O operations in terms of whole chunks. However, in the general case, one should choose an average chunkshape, e.g. the one shape that PyTables chooses for you. This is also the case in the setting of genomic analysis, which should be efficient in any scenario.

8.4 Weaknesses

8.4.1 Result irregularities

The results from the previous chapter, that have been discussed in this chapter, all come from running the *Operation performance tool* on our HyperBrowser instance (Appendix B) on the *insilico* server. Throughout the testing period we experienced some inconsistent results. Sometimes results varied by multiple factors. An example of this behavior is seen in Appendix C, section 3.1.

The insilico server is used by many users simultaneously. While the physical hardware on the server should be able to handle this, the reason for our inconsistent results may still be a consequence of not enough CPU capacity. A more plausible cause is that because of the heavy disk I/O of the operations, there was some times need to wait for I/O operations of other processes.

When we made sure to only run one test at a time, and when there was few other processes being run on the server, the results seemed to stabilize. There is, however, a risk that some of the results still is a result of these irregularities.

Other weaknesses

As briefly mentioned earlier, the track used for the test cases that involved *weights*, i.e. *Inter- and intrachromosomal*, was too small and had too few track elements. Hence, the two test cases where this track was involved did not give good, conclusive results, and may therefore not be thoroughly tested.

In addition, because of the irregularities with the results, another weakness of the results may be that in the 'Average of N runs' tests we should have used more than 10 runs to get more accurate results.

Another weakness of this work is the lack of isolated tests. While the test cases used to compare the two GTrackCore versions is meant to test individual areas of GTrackCore, they always perform multiple tasks. For instance, we never read contiguous data without first loading a track view. Because of this, we have had to rely on the detailed profiles to investigate the the actual run times of the basic operations. This may not be the appropriate method for such investigations as they might be imprecise due to the overhead of profiling.

8.5 How is data retrieval affected by the use of PyTables in GTrackCore?

The previous sections have all discussed the results from the last chapter. An important part of the investigating of whether PyTables is suitable in GTrackCore is the performance of data retrieval with the new data model. In this section we will sum the results that have been discussed up until now to try to answer the research question: *How is data retrieval in GTrackCore*

affected by the use of HDF5 and PyTables as opposed to the use of memmaps regarding performance?

Memmap compared to PyTables

- It is overall faster to read contiguous data from memmap files than when using PyTables and HDF5. This is probably because memmaps are a direct representation of NumPy arrays on disk, and is hence stored contiguously, while PyTables stores datasets in an HDF5 file in chunks, and must therefore read data in multiple disk read operations.

- Using the row iterator of tables in PyTables to iterate the track elements of a track view is about 10 times faster than the method used in the memmap solution. The speed difference is likely due to chunk caching in the HDF5 library, and an optimized row iterator in PyTables written in Cython.

- The track view loader in the PyTables version is slightly faster than the one in the memmap version. This mostly is because the sequential search to find the exact start and end indices in the memmap version is relatively slow, making the the binary search algorithm in the PyTables version faster in total.

Variants of PyTables

- Reading contiguous data from arrays is definitely faster than reading from tables columns. This is because one does not have the overhead of retrieving data you do not need, i.e. from the other columns in the table. Moreover, fewer chunks needs to be read, and therefore also fewer disk reads is required, when using arrays as more data can be put into each chunk.

- Compression of data has mostly a negative impact on performance when reading contiguous data. This is likely because the decompression takes longer time than what is saved by reading less data.

- While it has not been tested, using arrays in the track view iterator is probably slower than using a table because one would have to read from multiple datasets. However, the binary search algorithm could, in theory, slightly increase performance by using arrays, as opposed to tables, as one would read less data in total, though it is not tested.

8.6 Issue with indexing in PyTables

One of the reasons for using PyTables in the new version of GTrackCore was the ability in PyTables to created indices for some of the columns of genomic track. These indexed columns would be used to maybe ensure quickly searches for the start and end indices in the track view loader, which is used almost every time one want to retrieve data from GTrackCore. This

section will therefore discuss the research question: *How can the built-in indexing functionality in PyTables be used to ensure efficient data retrieval in GTrackCore?*

As described in section 6.2 on page 57, using queries on indexed columns did not work as as hoped. The problem was that PyTables retrieves the indices used for a query from disk for each query that is performed, with the `get_chunkmap` method. While this normally not is the most time consuming part of a single query, at least not for complex queries, e.g. involving many columns, it adds up to a lot when performing many queries. We, however, performed many simple queries that did not take much time each, but since the indices had to be read each time, the total time for all the queries was very high. The root of the problem seems to be that PyTables do not cache these indices. It may be more normal to only perform a single query within a given time period, as opposed to many queries like we do in GTrackCore. This may be why PyTables do not perform any caching of the indices. However, an issue posted on the PyTables repository on GitHub addresses the same problem. Here, one of the developers of PyTables suggests that "this should be called once and cached somewhere"[1].

If this issue is fixed by PyTables, and the indices are cached, there may be good reason to use queries instead of the ad-hoc binary search method. Moreover, having the ability to use indices will allow for faster retrieval of bounding regions too. As mentioned, in the search for an enclosing bounding region we query a non-indexed table. Currently, this is not a large issue as there normally are relatively few bounding regions in genomic tracks.

8.7 Utilizing PyTables

In this section we discuss whether there are any strengths of PyTables that can be utilized by GTrackCore. This is related to the research question: *Is GTrackCore facilitated to utilize the strengths of PyTables, regarding indexing and querying of tables?*

As we have discussed, querying indexed data in the track view loader of GTrackCore would probably help to ensure efficient data retrieval. However, because of the lack of caching of indices in PyTables, we have not been able to utilize queries in this setting. Hence, GTrackCore is currently not facilitated to utilize this feature of PyTables. There may, however, still be use for indexing and queries in other parts of GTrackCore if we were to extend its functionality, e.g. by creating a new data interface.

8.7.1 A new interface for the track data

To utilize more features of PyTables a possibility is to create a new interface against the track data, other than the current track view and graph view. The new interface would be centered around performing queries directly

[1] Issue regarding `get_chunkmap`: https://github.com/PyTables/PyTables/issues/187 (visited on 07/25/2014)

on the track data. Here, one can make queries for common operations, and retrieve results quickly when having indexed data. This would be an alternative to the two current main retrieval methods, i.e. the track view iterator and the 'as NumPy array' methods. Moreover, one can utilize the `Expr` class[1] of PyTables to perform algebraic operations on potentially large datasets, using PyTables as a computing kernel[2]. Using this class, one can evaluate expressions containing array-like objects, such as arrays or table columns in PyTables, more efficiently than when having to load them into memory separately. We have, however, not investigated the need and exact purpose for such operations in GTrackCore in this thesis.

8.7.2 Using the row iterator

As we have covered, tables in PyTables include a fast row iterator, which we utilize to iterate the track elements of a track much faster than in the old version. While this is not related to indexing or querying, it is still an important feature of the PyTables library that we currently utilize.

There may be many track operations for biological analysis where the result can be a new track, e.g. the union of two tracks. Here, one have to use the iterator to get each track element. For this type of operations it is beneficial to have an efficient track view iterator to yield track elements. Because the use of PyTables has increased the performance of the track view iterator, it might help make GTrackCore more attractive if it were to be a stand-alone command-line tool, equal to BEDTools.

8.8 Is PyTables and HDF5 suited as a new data model in GTrackCore?

In this section we briefly discuss the main research question, focusing on the retrieval part of GTrackCore: *Is the PyTables library, making use of the HDF5 format, a suitable replacement for NumPy memmaps in GTrackCore?* The question relates somewhat to the previous discussions in the sections, particularly with regards to performance.

Having the implementation of the PyTables version of GTrackCore as proof, we know that using PyTables do not limit any functionality of GTrackCore. All functionality available in the old version is also available in the new.

However, based on the discussed topics earlier on, we may indicate that the use of PyTables and HDF5 as a new data model in GTrackCore seems to be a less suitable option than NumPy memmaps when it comes to reading contiguous data. This is based on the assumption that the decrease in performance is excessive, i.e. too much decrease. We have seen that reading contiguous data from NumPy memmap files generally is faster than reading from HDF5 files. For very large biological analysis, where much data is read,

[1] Expr class: http://pytables.github.io/usersguide/libref/expr_class.html (visited on 07/25/2014)

[2] http://www.pytables.org/moin/ComputingKernel (visited on 07/25/2014)

this may be a problem. The 'Values (C)' test case, where over 20 GB of data is read, we saw that that the PyTables solution was about 80 % slower than the memmap solution. However, the 'Values (G)' test case, where less data in total was read, but about 28 000 track views were loaded, the PyTables version was faster than the memmap version. This was mostly because the track view loader, which is a large part of the data retrieval, is faster in the new version. The PyTables version had, however, much better performance for the track view iterator. We have seen that the row iterator for table datasets in PyTables has proven to be very efficient when iterating track elements of a track view, compared to using the memmap solution for the same task.

8.8.1 Discussing suitability

To compare the two GTrackCore versions we have created test cases that only resembles the statistical analysis tools in the Genomic HyperBrowser, and have not tested the use of PyTables in a real setting, i.e. in the actual statistical analysis tools. Hence, we do not know exactly how the GTrackCore is used and can not give a clear answer to whether the decrease in performance for reading contiguous data is significant or not. We can therefore not conclusively decide the suitability of using the PyTables library in GTrackCore without having even deeper knowledge of its usage.

Chapter 9

Conclusion

The goal of this work was the investigation of whether the PyTables library, making use of the HDF5 file format, is a suitable replacement as a new data model in GTrackCore, as opposed to using NumPy memmaps. This has been done by re-implementing large parts of GTrackCore to work with the new model, and then compared the old and new solution against each other.

Rongved [34] shows that using PyTables and HDF5 solves the issue of having a data model with multiple files for each genomic track, now using only one file. Furthermore, we learn that the use of PyTables has not decreased the performance of the preprocessor part of GTrackCore significantly, and is in some cases even faster.

Through performance testing of both versions of GTrackCore and deep analysis of these, we show that the use of the PyTables library and HDF5 decrease performance of data retrieval when reading contiguous data from disk compared to reading from NumPy memmaps. Reading data using the track view iterator is, however, much faster in the new version. Moreover, we have seen that using arrays as data source for reading contiguous data is faster than using table columns, and that compression of datasets mostly have a negative effect on retrieval.

Because the use of the PyTables library now makes it possible to store genomic tracks in a single file, while still having approximately the same performance as with the previous data model, we conclude that the use of PyTables and HDF5 as a new data model in GTrackCore seems to be a suitable alternative. This is based on the assumption that the decrease in performance for reading contiguous data not is a significant factor for biological analysis in the big picture.

Furthermore, we see that GTrackCore is not, as of now, facilitated to make use of the indexing and query features of the PyTables library. If caching of indices in PyTables is implemented, we can use queries to ensure quick loading of track view, and hence quick data retrieval. Another discussed option to utilize PyTables is to create a new interface to the track data in GTrackCore, which uses queries to perform common track operations quickly. In addition, compression of datasets in HDF5 may be used to decrease file sizes of genomic tracks, e.g. on home computers, but then with the consequence of having slightly slower retrieval of contiguous data.

9.1 Future work

Since the use of arrays i PyTables is faster when it comes to reading contiguous data compared to using tables, and because we currently are not using any features available only for tables, such as queries, one should investigate further the use of arrays. In particular the use of arrays in the track view iterator, while also in the track view loader when searching for the start and end indices of a track view.

In addition, there may be a more optimized chunkshape for the datasets stored in the HDF5 file than the one being automatically set by PyTables. The experiment presented in this work is not substantial enough, and should be further tested.

If one were to deem PyTables and HDF5 suitable in GTrackCore as it is now, one should also further explore the opportunities of utilizing queries in PyTables, such as the one proposed in this work.

Bibliography

Journal papers

[4] Kent Beck. 'Embracing change with extreme programming'. In: *Computer* 32.10 (1999), pp. 70–77.

[5] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries et al. 'Manifesto for agile software development'. In: (2001).

[19] Jeremy Goecks, Anton Nekrutenko, James Taylor, T Galaxy Team et al. 'Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences'. In: *Genome Biol* 11.8 (2010), R86.

[20] Sveinung Gundersen, Matúš Kalaš, Osman Abul, Arnoldo Frigessi, Eivind Hovig and Geir K Sandve. 'Identifying elemental genomic track types and representing them uniformly'. In: *BMC bioinformatics* 12.1 (2011), p. 494.

[21] Lawrence Hunter. 'Molecular biology for computer scientists'. In: *Artificial intelligence and molecular biology* (1993), pp. 1–46.

[35] Geir K Sandve, Sveinung Gundersen, Morten Johansen, Ingrid K Glad, Krishanthi Gunathasan, Lars Holden, Marit Holden, Knut Liestøl, Ståle Nygård, Vegard Nygaard et al. 'The Genomic HyperBrowser: an analysis web server for genome-scale data'. In: *Nucleic acids research* (2013).

[36] Geir K Sandve, Sveinung Gundersen, Halfdan Rydbeck, Ingrid K Glad, Lars Holden, Marit Holden, Knut Liestøl, Trevor Clancy, Egil Ferkingstad, Morten Johansen et al. 'The Genomic HyperBrowser: inferential genomics at the sequence level'. In: *Genome biology* 11.12 (2010), R121.

[46] Stefan Van Der Walt, S Chris Colbert and Gael Varoquaux. 'The NumPy array: a structure for efficient numerical computation'. In: *Computing in Science & Engineering* 13.2 (2011), pp. 22–30.

[47] Laurie Williams, Ron Jeffries, Robert R Kessler and Ward Cunningham. 'Strengthening the case for pair programming'. In: *IEEE software* 17.4 (2000), pp. 19–25.

Other written references

[23] Brian W Kernighan, Dennis M Ritchie and Per Ejeklint. *The C programming language*. Vol. 2. prentice-Hall Englewood Cliffs, 1988.

[34] Brynjar G. Rongved. 'Storage of Genomic Data using PyTables'. University of Oslo, to be submitted August 2014.

Online references

[1] Francesc Alted, Ivan Vilata et al. *PyTables: Hierarchical Datasets in Python*. 2002–. URL: http://www.pytables.org/ (visited on 06/19/2014).

[2] *Appendix C. PyTables' parameter files*. URL: http://www.pytables.org/docs/manual-2.2.1/apc.html (visited on 06/24/2014).

[3] *Assumptive documentation*. URL: http://invitro.titan.uio.no/wiki/hyperbrowser/doku.php/communicating_code_assumptions_through_source_code_documentation (visited on 05/08/2014).

[6] *BEDTools Documentation*. URL: http://bedtools.readthedocs.org (visited on 05/05/2014).

[7] *Biopython*. URL: http://biopython.org (visited on 05/05/2014).

[8] *Blosc compression*. URL: http://www.blosc.org/ (visited on 06/29/2014).

[9] *Camel case*. URL: http://en.wikipedia.org/wiki/CamelCase (visited on 05/06/2014).

[10] *Chunking in HDF5*. URL: http://www.hdfgroup.org/HDF5/doc/Advanced/Chunking/ (visited on 07/13/2014).

[11] *cProfile Module Reference*. URL: https://docs.python.org/2/library/profile.html#module-cProfile (visited on 07/11/2014).

[12] *Cython*. URL: http://www.cython.org/ (visited on 05/05/2014).

[13] *Galaxy Data Formats*. URL: https://usegalaxy.org/static/formatHelp.html (visited on 05/05/2014).

[14] *Genome Coordinate Conventions*. URL: http://alternateallele.blogspot.no/2012/03/genome-coordinate-conventions.html (visited on 06/03/2014).

[15] *Genomic Feature*. URL: http://genomevolution.org/wiki/index.php/Genomic_features (visited on 05/05/2014).

[16] *GetGalaxy*. URL: https://wiki.galaxyproject.org/Admin/GetGalaxy (visited on 05/05/2014).

[17] *Git*. URL: http://git-scm.com/ (visited on 05/08/2014).

[18] *GitHub*. URL: https://github.com/ (visited on 05/08/2014).

[22] *Interval Operations in Galaxy*. URL: http://wiki.galaxyproject.org/Learn/Interval%20Operations (visited on 05/05/2014).

[24] *Memmap*. URL: http://docs.scipy.org/doc/numpy/reference/generated/numpy.memmap.html (visited on 05/05/2014).

[25] *nose – Documentation*. URL: https://nose.readthedocs.org/en/latest/ (visited on 06/25/2014).

[26] *Numexpr*. URL: https://github.com/pydata/numexpr (visited on 06/29/2014).

[27] *NumPy*. URL: http://docs.scipy.org/doc/numpy/user/whatisnumpy.html (visited on 05/05/2014).

[28] *OPSI: The indexing system of PyTables*. URL: http://www.pytables.org/docs/OPSI-indexes.pdf (visited on 06/27/2014).

[29] *Optimization tips in PyTables*. URL: http://pytables.github.io/usersguide/optimization.html (visited on 07/13/2014).

[30] *PEP 8 – Style Guide for Python Code*. URL: http://legacy.python.org/dev/peps/pep-0008/ (visited on 05/06/2014).

[31] *PyTables Natrual Naming*. URL: http://www.pytables.org/moin/NaturalNaming (visited on 05/13/2014).

[32] *Python*. URL: https://www.python.org/ (visited on 05/05/2014).

[33] *Python Code Style*. URL: http://docs.python-guide.org/en/latest/writing/style/ (visited on 06/25/2014).

[37] *SciPy*. URL: http://docs.scipy.org (visited on 05/05/2014).

[38] *shelve – Python object persistence*. URL: http://docs.python.org/2/library/shelve.html (visited on 05/07/2014).

[39] *Snake case*. URL: http://en.wikipedia.org/wiki/Snake_case (visited on 05/06/2014).

[40] The HDF Group. *HDF5 User's Guide*. URL: http://www.hdfgroup.org/HDF5/doc/UG/index.html (visited on 05/05/2014).

[41] The HDF Group. *Hierarchical Data Format, version 5*. 1997-2014. URL: http://www.hdfgroup.org/HDF5/ (visited on 05/05/2014).

[42] *The Python Profilers*. URL: https://docs.python.org/2/library/profile.html (visited on 07/08/2014).

[43] *UCSC Data File Formats*. URL: http://genome.ucsc.edu/FAQ/FAQformat.html (visited on 05/05/2014).

[44] *UCSC: Genomic Coordinates*. URL: http://genomewiki.ucsc.edu/index.php/Coordinate_Transforms (visited on 05/05/2014).

[45] *UNIX File system*. URL: http://www.cis.rit.edu/class/simg211/unixintro/Filesystem.html (visited on 06/18/2014).

[48] *zlib compression*. URL: http://www.zlib.net/ (visited on 06/29/2014).

Appendices

Appendix A

PyTables version of GTrackCore

The Github repository for the PyTables version of the GTrackCore is available here: https://github.com/brynjagr/gtrackcore

Appendix B

GTrackCore instance of HyperBrowser

An instance of HyperBrowser for testing and comparison of GTrackCore, with the test tools for performance testing and profiling, is accessible here: https://hyperbrowser.uio.no/gtrackcore/

The two tools are available from the left-side menu, under the category "GTrackCore".

Appendix C

Supplementary material

All raw data used for results in this work, and a description of the tool used to generate the data, is available from a Galaxy page on the GTrackCore instance of HyperBrowser (Appendix B): https://hyperbrowser.uio.no/gtrackcore/u/henrik/p/retrieval-of-genomic-data-using-pytables

Appendix D

Automatic vs. large chunkshape

```
In [1]: import tables
In [2]: h1 = tables.open_file('repeating_elements.h5') # automatic
    chunkshape
In [3]: h2 = tables.open_file('repeating_elements_chunkshape_200000.h5')

In [4]: table1 =
    h1.root.hg19.sequence.repeating_elements.no_overlaps.repeating_elements
In [5]: table2 =
    h2.root.hg19.sequence.repeating_elements.no_overlaps.repeating_elements

In [6]: table1
Out[6]:
/hg19/sequence/repeating_elements/no_overlaps/repeating_elements
    (Table(5109922,)) 'repeating_elements'
  description := {
  "end": Int32Col(shape=(), dflt=0, pos=0),
  "score": StringCol(itemsize=2, shape=(), dflt='', pos=1),
  "start": Int32Col(shape=(), dflt=0, pos=2),
  "strand": Int8Col(shape=(), dflt=0, pos=3),
  "val": StringCol(itemsize=129, shape=(), dflt='', pos=4)}
  byteorder := 'little'
  chunkshape := (1872,)

In [7]: table2
Out[7]:
/hg19/sequence/repeating_elements/no_overlaps/repeating_elements
    (Table(5109922,)) 'repeating_elements'
  description := {
  "end": Int32Col(shape=(), dflt=0, pos=0),
  "score": StringCol(itemsize=2, shape=(), dflt='', pos=1),
  "start": Int32Col(shape=(), dflt=0, pos=2),
  "strand": Int8Col(shape=(), dflt=0, pos=3),
  "val": StringCol(itemsize=129, shape=(), dflt='', pos=4)}
  byteorder := 'little'
  chunkshape := (200000,)

In [8]: %timeit -n 10 for row in table1: pass
10 loops, best of 3: 673 ms per loop
```

```
In [9]: %timeit -n 10 for row in table2: pass
10 loops, best of 3: 464 ms per loop

In [10]: %timeit -n 10 table1.cols.start[:]
10 loops, best of 3: 627 ms per loop

In [11]: %timeit -n 10 table2.cols.start[:]
10 loops, best of 3: 517 ms per loop

In [12]: %timeit -n 10 table1.cols.val[:]
10 loops, best of 3: 985 ms per loop

In [13]: %timeit -n 10 table2.cols.val[:]
10 loops, best of 3: 813 ms per loop

In [14]: %timeit -n 10 table1.cols.val[:1000]
10 loops, best of 3: 396 µs per loop

In [15]: %timeit -n 10 table2.cols.val[:1000]
10 loops, best of 3: 14 ms per loop

In [16]: %timeit -n 10 table1.cols.val[:50000]
10 loops, best of 3: 9.14 ms per loop

In [17]: %timeit -n 10 table2.cols.val[:50000]
10 loops, best of 3: 18.8 ms per loop

In [18]: %timeit -n 10 table1.cols.val[:200000]
10 loops, best of 3: 33.8 ms per loop

In [19]: %timeit -n 10 table2.cols.val[:200000]
10 loops, best of 3: 27.6 ms per loop

In [20]: timeit -n 1000 table1.cols.val[2000000]
1000 loops, best of 3: 339 µs per loop

In [21]: timeit -n 1000 table2.cols.val[2000000]
1000 loops, best of 3: 14 ms per loop
```